CSEN 202
Introduction to Computer Programming

Lecture 3:
Decisions

Prof. Dr. Slim Abdennadher and
Dr Mohammed Abdel Megeed Salem,
slim.abdennadher@guc.edu.eg

German University Cairo, Department of Media Engineering and Technology

February 17/22, 2018

1 Synopsis

1.1 What you learned from lecture 2

Datatypes

- Datatypes are necessary to *capture properties* of data inside a *generic storage*. Strict typing prevents *errors*.
- Java offers primitive types:
  - Integral numeric types: **byte**, **short**, **int**, **long**
  - Floating point types: **float**, **double**
  - Other types: **boolean**, **char**
- The literals in Java are *also* typed
  - `3459` is of type **int**, `3459L` is of type **long**.
- Types can me *changed* by means of an *implicit* or *explicit* type cast:
  - `long l = 7364;`
  - `short s = (short) 7364;`

  The type-casting rules in Java are very *restrictive*.

Expressions

- Java expressions entail some basic *operations* (`+`, `-`, `*`, `/`, `%`, `==`, `!=`, `>`, `<`, `>=`, `<=`, `&&`, `||`, `!`,`&`, `|`, `^`, `~`)
- Assignments and even declarations return a *value*
  - `a = (b = 5);`
  - `float x = (float) (double d = .007);`
- Not mentioned in last lecture (but in the tutorials):
  - Pre- and post-increment/decrement
    `int j = ++i;`
    `int j = i--;`
  - Shortcut update operators (`+=`, `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`)
    `i += 2 * j + x;`
    `a %= b;`
  - Signed and unsigned Bit-shifts (`<<`, `>>`, `>>>`)
    `a >>> 4;`

Expressions

Typing overview

Java operators	Operands	Result
`+`, `-`, `*`, `/`, `%`	Numeric types	Numeric type
`==`, `!=`, `>`, `<`, `>=`, `<=`	**`boolean`**, **`char`**, or numeric types	**`boolean`**
`&&`, `||`, `!`	**`boolean`**	**`boolean`**
`&`, `|`, `^`, `~`	**`boolean`**, **`char`**, or integral numeric types	**`boolean`** or integral numeric type

Quirks

- The type **`boolean`** can *not* be casted in any way
- Do *not* confuse "`==`" and "`=`"!

2 Some basics

2.1 Structuring the code

Blocks

A note about structuring the code:

- Multiple statements can be summarized as a single *block* by enclosing them in *braces* "`{...}`".

  Example 1. ¡3-¿

```
int a = 5;
{                       ← opening brace
    int b = 5;
    a += b;
}                       ← closing brace
```

The statements `int b = 5;` and `a += b;` together form *one block*. Note the *indentation*

Blocks

Properties of blocks (`{...}`):

- A block can replace a *single* statement
- Variables that are *declared* inside a block can only be *used* inside that block

3 Control structures

Today's topic

control structures

Overview

Control structures influence the *execution of statements*
Two basic types:

- Conditional statements *execute* parts of the program *only* if some *condition* is met.
  - `if`-statement,
  - `switch`-statement,
  - conditional expression.
- Loop statements *execute* parts of the program *multiple times*
  - `while`-loop,
  - `do`-loop,
  - `for`-loop

3.1 Conditional statements

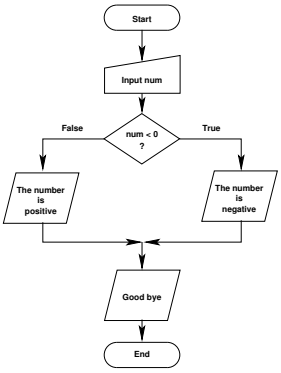
The `if`-statement

The form of the `if`-statement

```
if (condition)
    single statement or block for the true case
else
    single statement or block for the false case
```

- The condition must be an *expression* of type **`boolean`**
- The first statement or block (**`true`**-case) is *only* executed *if* the condition evaluates to **`true`**
- *Otherwise*, the second statement or block is executed
- The **`else`**-part is *optional*

Conditional operation – diagram




```
if ( num < 0 )
    System.out.println("The number " + num + " is negative");
else
    System.out.println("The number " + num + " is positive");
System.out.println("Good-bye for now");
```

Example

Number of students

```
System.out.print(numberOfStudents);
if (numberOfStudents == 1)
    System.out.print("student");
else
    System.out.print("students");
System.out.println(" registered.");
```

- If the number of students is 1, the code prints:

  `1 student registered`

- Otherwise (*e. g.*, `numberOfStudents` $= 5$) it prints the number of students:

  `5 students registered`

Blocks and braces

Is the following section of a program *correct*?

```
if ( num < 0 )
    System.out.println("The␣number␣" + num + "␣is␣negative");
else
    System.out.println("The␣number␣" + num + "␣is␣positive");
    System.out.print ("positive␣numbers␣are␣greater␣");
    System.out.println("or␣equal␣to␣zero␣");
System.out.println("Good-bye␣for␣now");
```

```
if ( num < 0 )
    System.out.println("The␣number␣" + num + "␣is␣negative");
else {              ← opening brace
    System.out.println("The␣number␣" + num + "␣is␣positive");
    System.out.print ("positive␣numbers␣are␣greater␣");
    System.out.println("or␣equal␣to␣zero␣");
}                   ← closing brace
System.out.println("Good-bye␣for␣now");
```

No. The programmer probably wants the three statements after the else to be part of a block, but has not used *braces* to show this. Correcting the program...

Boolean expressions within the condition

- Boolean conjunction: `&&`

```
// check that there are enough of both ingredients
if ( flour >= 4 && sugar >= 2 )
    System.out.println("Enough␣for␣cookies!" );
else
    System.out.println("sorry...." );
```

- Boolean disjunction: `||`

```
// check that at least one qualification is met
if ( cash >= 25000 || credit >= 25000 )
    System.out.println("Enough␣to␣buy␣this␣car!" );
else
    System.out.println("What␣about␣a␣Yugo?" );
```

- ...

Boolean expressions within the condition

- Boolean Negation: `!`

```
if ( !( speed > 2000 && memory > 512 ) )
    System.out.println("Reject␣this␣computer");
else
    System.out.println("Acceptable␣computer");
```

Nested `if`

Consider:

```
if (condition 1) if (condition 2) statement 1
else statement 2
```

- The ambiguous `else` is called "*dangling* `else`"
- In Java, an `else` belongs to the *closest* `if`
- Avoid this situation by using *braces*

```
if (condition 1)                    if (condition 1) {
    if (condition 2)                    if (condition 2)
        statement 1                         statement 1
    else                            else
        statement 2                         statement 2
                                    }
```

Refactoring `if`-conditionals

Refactoring	Before	After	Comment
Swap branches	`if (!condition) { statement 1 } else { statement 2 }`	`if (condition) { statement 2 } else { statement 1 }`	*Equivalent*
Remove redundant tests	`if (condition) { statement 1 } if (condition) { statement 2 }`	`if (condition) { statement 1 statement 2 }`	*Not* generally equivalent (why?)

Refactoring `if`-conditionals

Refactoring	Before	After	Comment
Extract to front	`if (condition) { *statement 1* statement 2 } else { *statement 1* statement 3 }`	*statement 1* `if (condition) { statement 2 } else { statement 3 }`	*Not* generally equivalent (why?)
Extract to back	`if (condition) { statement 1 *statement 3* } else { statement 2 *statement 3* }`	`if (condition) { statement 1 } else { statement 2 }` *statement 3*	*Equivalent*

Counterexample

Consider

```
if (i%7 != 0) {
    i++;
    j = i;
} else {
    i++;
}
```

This is *not* equivalent to

```
i++;
if (i%7 != 0) {
    j = i;
}
```

3.2 Conditional expression

Conditional operator

condition ? expression 1 : expression 2

- Operand types:
  - condition: **boolean**
  - expression 1 and expression 2: *Any* type
- Works like the **if**-statement, but for *expressions*
  - If the condition is **true**, the value of the whole expression is that of expression 1, otherwise that of expression 2.

Example 2. ¡4-¿

```
if (num < 0)
    x = -num;
else
    x = num;
```

is *equivalent* to `x = num < 0 ? -num : num;`

Example

```
// print the number of books found
public class Books
{
    int num = 4;
    public static void main(String[] args) {
        System.out.println("Number␣of␣hits:" + num + "␣" +
                           ((num == 1) ? "book" : "books")
                           );
    }
}
```

- Useful when duplication of code or the introduction of a variable can be *avoided*.

3.3 Multiple choice conditional

The **switch** statement

- Instead of using *multiple* **if**-then-**else** branches which test a single value against several constants, the **switch**-statement can be used.

```
switch (expression)
    {
    case literal 1: statements; break;
    case literal 2: statements; break;
    ...
    case literal n: statements; break;
    default: statements;
    }
```

The `switch` statement

```
switch (expression)
    {
    case literal 1: statements; break;
    case literal 2: statements; break;
    ...
    case literal n: statements; break;
    default: statements;
    }
```

- If one case branch *matches*, *all* statements after it will be executed. Use **`break`** to control this
- Otherwise, the statements after the (optional) **`default`** are executed.

Example I

```
switch(studentsNr)
    {
    case 0:
        System.out.print("no one");
    case 1:
        System.out.print("1 student");
    default:
        System.out.print(studentsNr);
        System.out.print(" students");
    }
System.out.println(" registered");
```

```
switch(studentsNr)
    {
    case 0:
        System.out.print("no one"); break;      ← break added
    case 1:
        System.out.print("1 student"); break;  ← break added
    default:
        System.out.print(studentsNr);
        System.out.print(" students");
    }
System.out.println(" registered");
```

- Will this work as intended? *No!*
- **`break`**-statements need to be added to terminate the branches.

Example II

Problem:
Display the name of the month, based on the value of month, using the **switch**-statement

Example II

```
int month = 8;
switch (month) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    case 4: System.out.println("April"); break;
    case 5: System.out.println("May"); break;
    case 6: System.out.println("June"); break;
    case 7: System.out.println("July"); break;
    case 8: System.out.println("August"); break;
    case 9: System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
    default: System.out.println("There␣is␣no␣such␣month!");
    }
```

Comparison

Switch statement

```
switch(studentsNr) {
  case 0:
    System.out.print("no␣one");
    break;
  case 1:
    System.out.print("1␣student");
    break;
  default:
    System.out.print(studentsNr);
    System.out.print("␣students");
  }
System.out.println("␣registered");
```

If statement

```
if (studentsNr == 0)
  System.out.print("no␣one");
else if (studentsNr == 1)
  System.out.print("1␣student");
else {
  System.out.print(studentsNr);
  System.out.print("␣students");
}
System.out.println("␣registered");
```

Remarks

Notes about the switch statement

- Advantage: All branches test the *same value* (in the example: `studentsNr`)
- The test cases must be *integers*, or **char** or *strings*. You *cannot* use a **switch** to branch on floating-point or boolean values.

  The following fragment of code will produce an *error*:

  ```
  switch(studentGPA) {
    case 1.3 : System.out.println("Excellent"); break;
    case 4.0 : System.out.println("Sufficient"); break;
    case 2.0 : System.out.println("Good"); break;
    ...
  }
  ```

4 Coming up

4.1 Next week

Next week's topics

- *Iterative* constructs in Java