Module material (temporary location)

- Lab groups
- Lab ticketing system sign-up
- Shared dropbox folder. Check for link on moodle.

GC01 Introductory Programming

Induction Week – Lectures 2 and 3

Dr Ghita Kouadri Mostefaoui
Department of Computer Science

Outline

- Variables
- Interactive programs: User input
- Java libraries
- Boolean type and expressions
- Control flow: If-else statement
- Strings
- Java comments and Javadoc

Deadlines for tutorial worksheets

- Deadline for tick-off for induction week tutorial is ***5 October.***

- Subsequently, from week 1, the tick off will be due the following week (***during your official lab session***).

Why Java?

- It's the current "hot" language
- It's object-oriented
- It has a vast library of predefined objects and operations
- It's platform independent
  - this makes it great for web programming
- It's more secure

Variables

"*Once a programmer has understood the use of variables, he has understood the essence of programming*"

-Edsger Dijkstra

Variables

- **variable**: A piece of your computer's memory that is given a name and type, and can store a value.
  - We use variables to store the results of a computation and use those results later in our program.

- Variables are a bit like the 6 preset stations on your car stereo, except we can, essentially, have as many of them as we want, and we give them names, not numbers.

Declaring variables

- **variable declaration statement**: A Java statement that creates a new variable of a given type.
  - A variable is *declared* by writing a statement that says its type, and then its name. (The name is an *identifier*.)

- Declaration statement syntax:

  <type> <name> ;
  - Example: `int x;`
  - Example: `double myGPA;`

- It is also legal to declare multiple variables of the same type on one line:

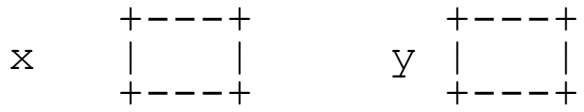
  <type> <name>, <name>, ..., <name> ;
  - Example: `int a, b, c;`

More on declaring variables

- Declaring a variable sets aside a chunk of memory in which you can store a value.

```
int x;
int y;
```

  - A (crude) diagram of part of the computer's memory:



(The memory has no value in it yet.)

- The compiler will fail if you try to declare a variable twice, or declare two variables with the same name.
  - Illegal:

```
int x;
int x;   // variable x already exists!  ERROR
```

- When tracing code, draw boxes for variables!

Assignment statements

- **assignment statement**: A Java statement that stores a value into a variable's memory location.
  - Variables must be declared before they can be assigned a value.

- Assignment statement syntax:

  <name> `=` ***<value>*** `;`

  - Example: `x = 3;`
  - Example: `myGPA = 3.95;`

  - Another (crude) diagram of part of the computer's memory:

```
     +---+              +------+
x    | 3 |      myGPA   | 3.95 |
     +---+              +------+
```

  - Technically, `=` is an operator like `+` or `*`, called the *assignment operator*, with very low precedence (it is carried out last).

More about assignment

- The ***<value>*** assigned to a variable can be a complex expression. The expression will be evaluated, and the variable will store the result.
  - Example:

```
x = (2 + 7) * 3 * 5;
```

- A variable can be assigned a value more than once in the program.
  - Example (Draw the boxes!!):

```
int x;
x = 3;
System.out.println(x);    // 3

x = 4 + 7;
System.out.println(x);    // 11
```

Using variables' values

- Once a variable has been assigned a value, it can be used in an expression, just like a literal value.

```
int x;
x = 3;
System.out.println(x * 5 - 1);
```

  - The above has output equivalent to:

```
System.out.println(3 * 5 - 1);
```

- A variable needs to be initialized before use. Otherwise, it will either raise a warning, an error or simply produce the wrong output as some variables in java are given a default value.
  - Bad:

```
int x;
```

  - Good
  - `int x = 0;`

Assignment and types

- A variable can only store a value of its own type.
  - Illegal: `x = 2.5;  // ERROR: x can only store an int`
  - (Technically, the value does not need to be the same type as the variable--it can be any type that Java knows how to convert into the variable's type... see below.)
- An `int` value can be stored in a variable of type `double`. The value is converted into the equivalent real number.
  - Legal: `double myGPA = 4;`

```
          +-----+
myGPA     | 4.0 |
          +-----+
```

Practice: Assignment examples

- What is the output of the following Java code?

```
int number;
number = 2 + 3 * 4;
System.out.println(number - 1);

number = 16 * 6;
System.out.println(2 * number);
```

- What is the output of the following Java code?

```
double average;
average = (9 + 8) / 2;
System.out.println(average);

average = (average * 2 + 10 + 8) / 4;
System.out.println(average);
```

Practice: Assignment examples

- What is the output of the following Java code?

```
int number;
number = 2 + 3 * 4;
System.out.println(number - 1);
> 13
number = 16 * 6;
System.out.println(2 * number);
> 192
```

- What is the output of the following Java code?

```
double average;
average = (9 + 8) / 2;
System.out.println(average);
> 8.0
average = (average * 2 + 10 + 8) / 4;
System.out.println(average);
> 8.5
```

Homework

- Course textbook

  Chapter 5 – Operators – *Operator precedence*

Declaration and initialisation

- A variable can be declared and assigned an initial value in the same statement, to save lines in your program.

- Declaration and initialization statement syntax:

  <type> <name> `=` ***<value>*** `;`

  - Example: `double myGPA = 3.95;`
  - Example: `int x = (11 * 3) + 12;`

same effect as:

```
double myGPA;
myGPA = 3.95;

int x;
x = (11 * 3) + 12;
```

- It is also legal to declare/initialize several at once:

  <type> <name> `=` ***<value>*** `,` ***<name>*** `=` ***<value>*** `;`

  - Example: `int a = 2, b = 3, c = -4;`
  - Example: `double grade = 3.5, delta = 0.1;`

Interactive programs

- We have written programs that print console output, but it is also possible to read *input* from the console.
  - The user types input into the console. We capture the input and use it in our program.
  - Such a program is called an *interactive program*.

- Interactive programs can be challenging.
  - Computers and users think in very different ways.
  - Users misbehave.

Input and `System.in`

- `System.out`
  - An object with methods named `println` and `print`
- `System.in`
  - not intended to be used directly
  - We use a second object, from a class `Scanner`, to help us.

- Constructing a `Scanner` object to read console input:

  ```
  Scanner name = new Scanner(System.in);
  ```

  - Example:

  ```
  Scanner console = new Scanner(System.in);
  ```

Java class libraries, import

- **Java class libraries**: Classes included with Java's JDK.
  - organized into groups named *packages*
  - To use a package, put an *import declaration* in your program.

- Syntax:

```
// put this at the very top of your program
import packageName.*;
```

- `Scanner` is in a package named `java.util`

```
import java.util.*;
```

  - To use `Scanner`, you must place the above line at the top of your program (before the `public class` header).

Example `Scanner` usage

```
import java.util.*;   // so that I can use Scanner

public class ReadSomeInput {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("How old are you? ");
        int age = console.nextInt();

        System.out.println(age + "... That's quite
  old!");
    }
}
```

- Output (user input underlined):

```
How old are you? 14
14... That's quite old!
```

`Scanner` methods

Method	Description
`nextInt()`	reads a token of user input as an `int`
`nextDouble()`	reads a token of user input as a `double`
`next()`	reads a token of user input as a `String`
`nextLine()`	reads a *line* of user input as a `String`

Each method waits until the user presses 'Enter'.

- The value typed is returned.

```
// prompt
System.out.print("How old are you? ");

int age = console.nextInt();

System.out.println("You'll be 40 in " +
                   (40 - age) + " years.");
```

- **prompt**: A message telling the user what input to type.

Another `Scanner` example

```
import java.util.*;   // so that I can use Scanner
public class ScannerSum {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Please type three numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();
        int num3 = console.nextInt();

        int sum = num1 + num2 + num3;
        System.out.println("The sum is " + sum);
    }
}
```

- Output (user input underlined):

```
Please type three numbers: 8 6 13
The sum is 27
```

  - The `Scanner` can read multiple values from one line.

Input tokens

- **token**: A unit of user input, as read by the `Scanner`.
  - Tokens are separated by *whitespace* (spaces, tabs, newlines).

- When a token is not the type you ask for, it crashes.

```
System.out.print("What is your age? ");
int age = console.nextInt();
```

Output:

```
What is your age? Timmy
java.util.InputMismatchException
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
      ...
```

Scanners as parameters

- If many methods read input, declare a `Scanner` in `main` and pass it to the others as a parameter.

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    int sum = readSum3(console);

    System.out.println("The sum is " + sum);
}

// Prompts for 3 numbers and returns their sum.
public static int readSum3(Scanner console) {
    System.out.print("Type 3 numbers: ");

    int num1 = console.nextInt();
    int num2 = console.nextInt();
    int num3 = console.nextInt();

    return num1 + num2 + num3;
}
```

Boolean Type and Expressions

Boolean values

- Named after George Boole (1815-1864), who invented logic and defined Boolean algebra.
- A variable of the primitive data type boolean can have two values: *true* and *false*.
- Boolean variables are used to indicate whether a condition is true or not, or to represent two states, such as a light being on or off.
- Examples:

```
boolean hasLicense= true;
boolean isDone = false;
boolean isTurnedOn;
```

Boolean expressions

- Arithmetic expressions use the operators

  + - * / % ++ --
- assignment expressions use the operator

  =
- Boolean expressions use *relational* and *logical* operators.
- The result of a Boolean expression is either *true* or *false*.
- Boolean expressions allow us to write programs that decide whether to execute some code or not.
- These decisions changes the flow of the program execution.

Relational operators

- Relational operators compare *two arithmetic expressions* and evaluate to a *boolean result*.

Operator	Name	Example	Value
==	Equal to	`1 + 1 == 2`	`true`
!=	Not equal to	`3.2 != 2.5`	`true`
<	Less than	`10 < 5`	`false`
<=	Less than or equal to	`2 <= 5`	`true`
>	Greater than	`100 > 126`	`false`
>=	Greater than or equal to	`5.0 >= 5.0`	`true`

Careful: Do not confuse the assignment operator = with the equality operator ==.

Practice

```
int x = 15;
int y = 100;
System.out.println(x > y);
System.out.println(x < 15);
System.out.println(x <= 15);
System.out.println(x == y);
System.out.println(x != 5);
System.out.println(x * -y > 0);
boolean isBigger = x > y;
```

Logical Operators

- Java provides logical operators.
  - The binary logical operators combine two boolean expressions into one.
  - The unary logical operator switches the value of a boolean expression.

Operator	Meaning	Kind	Example	Result
`&&`	AND	Binary	`(2 == 3) && (-1 < 5)`	false
`||`	OR	Binary	`(2 == 3) || (-1 < 5)`	true
!	NOT	Unary	`!(2 == 3)`	true

  - Binary logical operators have lower precedence than relational operators (they will be evaluated after)
  - Use parentheses.

Logical Operator Truth Tables

- Truth Tables show the result of a logical expression based on the values of the operands.

Op1	Op2	Op1 && Op2
true	true	true
true	false	false
false	true	false
false	false	false

Op1	Op2	Op1 || Op2
true	true	true
true	false	true
false	true	true
false	false	false

Op1	!Op1
true	false
false	true

Logical Operator Practice

- `2 > 3 && 4 < 5`
- `2 < 3 && 4 < 5`
- `2 > 3 || 4 < 5`
- `2 > 3 || 4 > 5`
- `!(2 > 3)`

Logical Operator Practice

- `2 > 3 && 4 < 5`
  `false` – first operand is `false`
- `2 < 3 && 4 < 5`
  `true`
- `2 > 3 || 4 < 5`
  `true`
- `2 > 3 || 4 > 5`
  `false` – both operands are `false`
- `!(2 > 3)`
  `true` – operand is `false`

Decision Structures: The If-Else Statement, Nested If Statements, and String Comparison

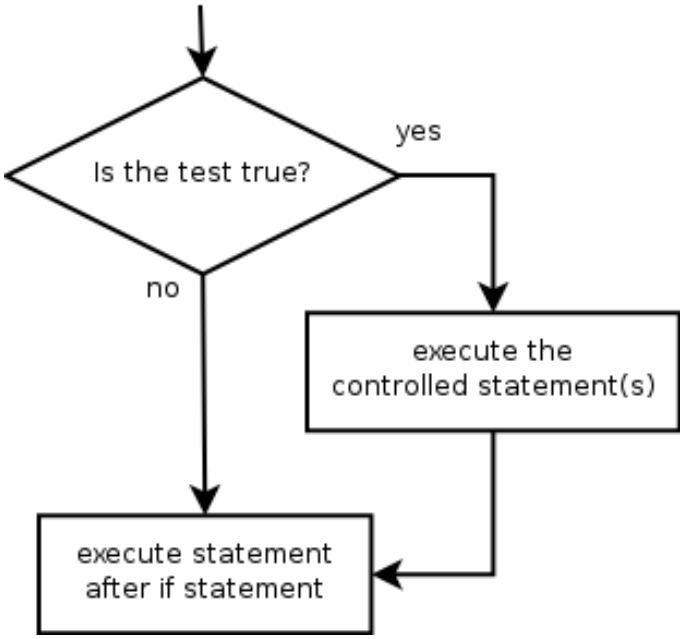
Decision statement

- In a decision structure's simplest form certain statements are executed only when…
  - a specific condition exists.
- It is said that the statements inside of the decision structure are…
  - conditionally executed.
- Relational Operators determine whether…
  - a specific relationship exist between two values.
- Some relational operators in Java are…
  - `>`, `<`, `>=`, `<=`, `==`, `!=`

`if` statement - General form

- General form of an `if` statement:

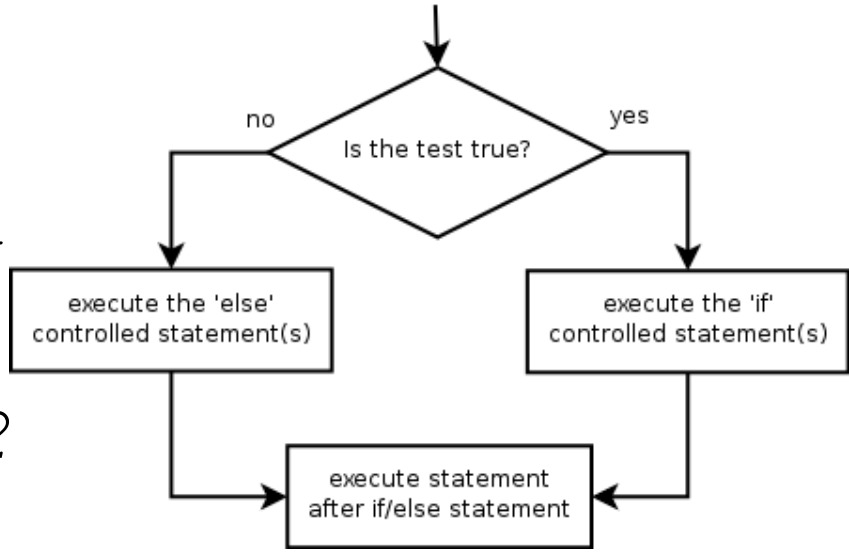
```
if(BooleanExpression)
   statement;

if(BooleanExpression) {
   statement1;
   statement2;
   ...
}
```



`if-else` - General form

- General form of an `if-else` statement:

```
if(BooleanExpression)
   statement or block 1
else
   statement or block 2
```



The `if-else if` Statement

- Sometimes you need to be able to test a series of conditions
  - You can do this with the if-else if statement
- General form:

```
if (BooleanExpression1)
    statement or block 1
else if (BooleanExpression2)
    statement or block 2
else
    statement or block 3
```

  - If *BooleanExpression1* is true, then *statement or block 1* is executed.
  - If *BooleanExpression1* is false, then *BooleanExpression2* is tested.
    - If *BooleanExpression2* is true, then *statement or block 2* is executed.
    - If *BooleanExpression2* is false, then *statement or block 3* is executed.
  - Note: You can have as many `if else` clauses as is needed.

Practice

```
public class IfElseDemo {

 public static void main(String[] args) {

        int testscore = 76;
        char grade;

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

Nested `if` Statements

- Nesting is enclosing one structure inside of another.
  - A block in Java can contain any valid Java code, this includes other `if` statements:

```
if(BooleanExpression1) {
    if(BooleanExpression2) {
        statement1;
        statement2;
    }
    statement3;
    statement4;
}
```

- If *BooleanExpression1* is `true` and *BooleanExpression2* is `true`, what is executed?
  - *statement1, statement2, statement3, statement4*
- If *BooleanExpression1* is `true` and *BooleanExpression2* is `false`, what is executed?
  - *statement3, statement4*
- If *BooleanExpression1* is `false`, what is executed?
  - Nothing

Practice

```
public class NestedIfDemo {

	public static void main(String[] args) {
		int courseValue = 100;
		String courseName = "Java";

		if(courseName == "Java")
		{
			if(courseValue < 100)
			{
				System.out.println("Value is with range");
			}
			else {
				System.out.println("Course is Expensive !!!!!!");
			}

			System.out.println("Decision made.");
		}

	}

}
```

Block-Level Scope

- If a variable is declared inside of a block, it is said to have block-level scope.
  - If a variable has Block-Level Scope it is in scope from its declaration to the ending of the block in which it was declared.

```
if(BooleanExpression) {
    int x;
    …
}
```

    - The variable `x` has scope from the point it was declared to the `}`
    - This does not have much use now, but will be more useful when we learn loops.

The Conditional Operator

- Java provides and operator to create short expressions that work like `if-else` statements.

BooleanExpression `?` *Value1* `:` *Value2*`;`

  - If *BooleanExpression* is `true`, *Value1* is returned
  - If *BooleanExpression* is `false`, *Value2* is returned

- Example:

```
if (score < 60)
    System.out.println("You Fail");
else
    System.out.println("You Pass");
```

Can be replaced by a one line code

```
System.out.println("You "
                   + (score < 60 ? "Fail" : "Pass"));
```

Practice

```
public class OneLinerMax {

    public static void main(String[] args) {
        int a = 15;
        int b = 30;
        int max;

        if (a > b) {
              max = a;
            }
            else {
              max = b;
            }
System.out.println("max = " +  max);
    }
}
```

Solution

```
public class OneLinerMax {

    public static void main(String[] args) {
        int a = 15;
        int b = 30;
        int max;

    max = (a > b) ? a : b;

    System.out.println("max = " +  max);
    }
}
```

Comparing `String` objects

Comparing `String` Objects

- Imagine you have declared two `String` variables as such:

```
String x = "Building";
String y = "apple";
```

- What does `x == y` resolve to?
  - `false`, but not for the reason you think!
    - The `==` operator compares what the reference values are (what objects they are pointing to), NOT what the value of the string is.
      - In some cases this causes problems
- You should use methods in the `String` class in order to compare `String` variables
  - `equals`
  - `compareTo`

String `equals` method

- To check if two strings are the same you can use the `equals` method.

`StringReference``.equals(`*`OtherString`*`);`

  - If the string referred to by *`StringReference`* is equal to the one referred to by *`OtherString`*, then `true` is returned.
    - Otherwise `false` is returned.
  - This comparison is case-sensitive (“`Building`” and ”`builDing`” are not equal)
    - To do case-insensitive comparison, use `equalsIgnoreCase()`

Homework

- `compareTo()` method

  https://www.youtube.com/watch?v=iTC43mLZG38

Java comments and Javadoc

Java comments

```
// This kind is to the end of the line
```

- for programmers who must read or modify your code.

```
/* This kind of comment can span multiple lines */
```

- for programmers who must read or modify your code.

```
/**
  * This kind of comment is a special
  *  'javadoc' style comment
  */
```

- for those who must use your code.

Javadoc Tool

- javadoc: The program to generate java code documentation.
- *Input*: Java source files (.java)
  - Individual source files
  - Root directory of the source files
- *Output*: HTML files documenting specification of java code
  - One file for each class defined
  - Package and overview files

Documentation with javadoc

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
* </p>
*
* @author  John Doe
* @version 2.1
* @since   1.0
*/
public class HelloWorld {
   public static void main(String[] args) {

        System.out.println("Hello World!");
   }
}
```

Example javadoc tags

Tag	Description	Syntax
@author	Adds the author of a class. *(classes and interfaces only)*	@author *name-text*
@version	Holds the current version number of the software that this code is part of. (*classes and interfaces only*)	@version *version-text*
@param	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section. *(methods and constructors only)*	@param *parameter-name description*
@return	Adds a "Returns" section with the description text. *(methods only)*	@return *description*

Demos and Links

- *Example 1*
  - http://www.tutorialspoint.com/java/java_documentation.htm
- *Example 2: Java™ Platform, Standard Edition 8 API Specification*
  http://docs.oracle.com/javase/8/docs/api/
- *Oracle "How to Write Doc Comments for the Javadoc Tool"*
  - http://www.oracle.com/technetwork/articles/java/index-137868.html