CSEN202 – Introduction to Computer Programming

Topics:

Welcome and Organization
Introduction to Java
Small Java Programs
How to Run Java Programs
Primitive Data Types
Expressions and Arithmetic

Prof. Dr. Slim Abdennadher

http://cs.guc.edu.eg/

6.3.2008

Course Structure

- **Lectures**: divided into three groups
- **Exercises and Homework**
  - Practical Assignments
  - Use feedback from tutors
- **Labs**
  - Supervised lab Assignments

WWW-page: Useful info and important announcements

`http://www.cs.guc.edu.eg/`

Why should you learn CSEN202?

- Improve your problem solving skills (clarity, precision, logic, ...)
- To use computers for problem solving
- Acquire new skills that will allow you to create useful and customized computer-based applications
- It is in the curriculum
- Acquire a useful vocabulary that will impress others in geeky conversations

Tentative Grading

Overall weighting for your grade

- **10%** for assignments
- **25%** for quizzes
- **25%** for mid-term exam
- **40%** for final exam

Tell me and I will forget;
show me and I may remember;
involve me and I will understand

Keep up with the course material

- Attend lectures, tutorials, and labs
- Participate in the discussions (be active)
- Solve the assignments and understand the model answers provided

Visit course home page regularly for announcements and supplemental material

`http://www.cs.guc.edu.eg/`

Problem Solving using a Programming Language

- A **programming language** specifies the words and symbols that we can use to write a program.
- A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid **program statements**.
- Examples of Programming Languages:
  - Fortran, Cobol, C++, C, Pascal, Prolog, **JAVA**

Course Outline

- Introduction to Java
- Fundamental Data Types
- Decisions
- Iteration
- Methods
- Recursion
- Objected-Oriented Programming: Classes and Objects
- Arrays
- Applets


Bandar Lampung
2. Jakarta
Java
Selat Sunda
Serang
1
Semarang
Surabaya
Madura
Selat Madura
Bandung
3
4
5
6
Yogyakarta
Denpasar
1. Banten
2. Jakarta Special Capital City District
3. Jawa Barat (West Java)
4. Jawa Tengah (Central Java)
5. Jawa Timur (East Java)
6. Yogyakarta Special Region

Java

Java

Origin of Java

- Began in 1991 with Green Team at Sun Microsystems in Menlo Park, CA
- Initial title was **OAK** (Object Application Kernel)
- The **initial goal** was the development of a programming language for embedded devices, e.g. toaster, coffee machine, VHS recoder, . . .
- **Java** created in 1992 by James Gosling, Patrick Naughton & Mike Sheridan
- Digital TV applications failed to generate business
- Focus turned to the **Internet**
- **New goal** was a **general purpose language with an emphasis on portability and interpretation**

History of Java

- **Java was released in 1995**
  - C functionality
  - Object Oriented (OO) capabilities
  - Other nice features (e.g. garbage collection)
- **Advantages:**
  - Simple for an OO language
  - Secure and reliable
  - platform independent: will work on any processor that has a Java interpreter – Java Virtual Machine
  - extensive libraries (esp. graphics & WWW)
- **Disadvantages:**
  - slower than C (more overhead)
  - limits user ability

The First Java Program

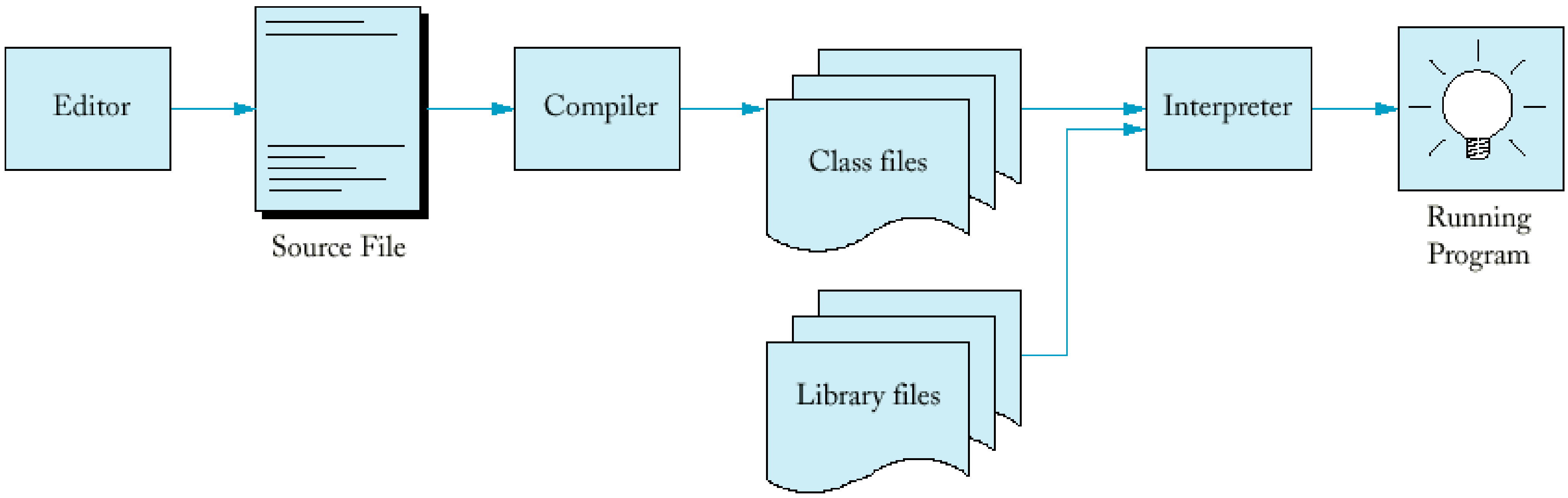
```
public class Hello
{
   public static void main(String[] args)

   {
        // display a greeting in the console window
        System.out.println("Hello, World!");
   }
}
```

- This code defines a **class** named `Hello`.
- The definition must be in a file `Hello.java`
- The **method** `main` is the code that runs when you execute the program

Building and Executing Java Code

- Source file name must end in `.java`
- Source file name must match the name of the public class
- **Java Development Kit (JDK)** must be installed to compile and run the programs
- **Compiling** to produce `.class` file
  - `javac Hello.java`
- **Running** in the JVM environment
  - `java Hello`
- Notice **the lack of `.class` extension**

From Source Code to Running Program



- The **Java compiler** does not produce machine code.
- The **Java compiler** produces **byte code** for the **Java Virtual Machine (JVM)**.
- The **JVM** for a platform **reads byte code** and **translates it to machine code at run time**.

Not important for this Lecture

```
public class Hello
{  public static void main(String[] args)
   {     // display a greeting in the console window
         System.out.println("Hello, World!");
   }
}
```

- `public class` *ClassName*: `public` denotes that the class is usable by the "public".
- `public static void main(String[] args)`: defines a method called `main`.
- The parameter `String[] args` contains the command line arguments
- The keyword `static` means that `main` does not inspect or change objects of the `Hello` class.
- The terminal window is represented in Java by an object called `out`.
- The `System` class contains useful objects and methods to access system resources.
- To use `out` object in the `System` class, we must refer to it as `System.out`.
- The `println` method will print a line of text.

Identifiers

- Names in programs are called **identifiers**.
- **Identifiers**
  - Always start with a letter.
  - Can include, digits, underscore and the dollar sign symbol.
  - Must be different from any Java reserved words (or keywords). Keywords that we have seen so far include: `public`, `static`, `class`, and `void`.
  - Are **case-sensitive**, for example `foobar`, `Foobar`, and `FOOBAR` are all different.
- We should try to use **descriptive names**.

Comments

To make our code understandable, we **comment** sections whose purpose is not immediately obvious.

- First kind of comments:

```
/* This is one kind of comment
   that can span several lines. Don't
   forget to put the closing
   characters at the end.
*/
```

- Second kind of comments

```
// This is the other type of comment.
// It covers the entire line
// and requires a new set
// of slashes for each new line.
```

Errors

- **Syntax errors:** Detected by the compiler
  - `System.ouch.print("Hello");`
  - `System.out.print("Hello);`
- **Logic errors:** Detected hopefully through testing
  - `System.out.print("Hell");`
- **Runtime errors:** Detected by the JVM
  - `System.out.print(1/0);`

Variables and Primitive Data Types

- A **variable** is a name for a location in memory.
- **Data Type Rules**
  - Must be declared with name and type
  - Value is optional
  - Cannot have two variables with same name and different type
  - **Naming convention**: Consecutive words, first letter of each word capitalized, except for first word, e.g. `numBuffalloWashed`
- **Java Data Types**: **Java literals** are of type
  - Boolean
  - Character
  - Integer
  - Floating point

Booleans

- **Boolean variables** can only take the values `true` or `false`. They are often used to test for conditions in a program.
- **Memory space**: 1 bit
- **Examples**:

```
boolean t = true;
boolean f = false;
```

Characters

- **Character variables** can store one character. A character value is a character surrounded by single quotes.

  `char p = ‘P’`

- All characters are represented by **16-bit unicode**. ’`\u`’ followed by four hexadecimal digits represent 16 bit unicode character.

  `char x = ’\u1234’`

- **Some special characters** are:

Escape Sequence	Unicode	Character
`\b`	`\u0008`	Backspace
`\n`	`\u000a`	Line feed
`\t`	`\u0009`	Horizontal Tabulation
`\’`	`\u0027`	Single quote
`\"`	`\u0022`	Double quote
\\	`\u0055`	Backslash

Integers

- An **integer** is any whole number, negative or positive, including 0.
- In Java, we can have integers of **different sizes**
  - `long` (**8 bytes**) can store values from -2^{63} to $2^{63} - 1$
  - `int`: (**4 bytes**) can store values from -2^{31} to $2^{31} - 1$
  - `short`: (**2 bytes**) can store values from -2^{15} to $2^{15} - 1$
  - `byte`: (**1 byte**) can store values from -128 to 127
- When using integers, we usually use `int`
- **Examples**:

```
byte b = 127;
short s = -32768;
int i = 4;
```

Integer literals

- An integer value, or **literal**, can be written in decimal, hexadecimal (base-16) or octal (base-8):
  - A **hex literal** starts with '`0x`', e.g.: `0x1f` $(= 31_{10})$
  - An **octal literal** starts with just '`0`', e.g. `072` $(= 58_{10})$
  - A **decimal literal** is just a regular number that does not start with '`0`', e.g.: `123`
- Integer literals are by **default** of type `int`.
- A `long` literal **ends with** `L`.

Integer Conversions (I)

If an `int` literal is small enough to fit into a `byte` or a `short`, it will be automatically converted. The same is true for `long` literals and `int`, `byte`, and `short`.

```
byte b = 0x7F;                /* 7 bits, OK */
short s = 0x7FFF;             /* 15 bits, OK */
int i = 0x12345678L;          /* 29 bits, OK */

byte b2 = 0xFF;               /* Error: 255 > 127 */
int i2 = 0x123456789ABCDEFL; /* Error: too big */
```

Integer Conversions II

- If a literal is too big for its target variable, you must explicitly convert it using a **type cast**. The number is converted by truncating the extra bits, which is probably not what you want.

  ```
  /* 0x100 = 256 */
  byte b = (byte) 0x100;
  /* b now equals 0! */
  ```

- An `int` literal can always be assigned to a `long` variable – its value will be the same as if it was assigned to `int` variable.

Floating-Point Numbers

- Floating-point numbers are used to represent reals, i.e. numbers that may have fractional parts, e.g. `123.4`, `55.`, `.99`
- The **Java floating-point types** are:
  - `float`: 32 bits
  - `double`: 64 bits
- A `float` literal ends with '`f`'.
- **Examples**:

```
float f = 123.4f
float f2 = .99f
```

- The floating point representation may use a form of **scientific notation** multiplying the literal by 10^n
- **Examples**:

```
float f = 1.234e2f
float f2 = 9.9e-1f
```

- A floating-point literal is by default of type `double`.

Floating-Point Conversions

- The only automatic conversion between floating-point types is the assignment of a `float` value to a `double`.

```
double d = 1.23F; /* OK */
float f = 5.99;  /* Error: Cannot assign double to float */
```

- When an integer literal is assigned to a floating-point type, it is automatically "promoted" to floating-point, even if that means a loss of precision.

```
float f = 2;                /*  OK, f = 2.0 */
float f2 = 1234512345L;     /*  OK, but f2 = 1234512384 */
```

Strings

- `String` is Not a primitive data type: It is an **Object**.
- Predefined class `String` has special support in Java.
- A string literal is surrounded by double quotes.

  ```
  String hamlet = "to be or not to be"
  ```

- Once a string has been created, we can use the **dot operator** to invoke its methods:

  ```
  hamlet.length();
  ```

- The `String` class has several methods to manipulate strings
  - `char charAt(int index)`: returns the character at the specified index
  - `String toLowerCase()`: Converts all of the characters in this String to lower case.
  - `String replace(char oldChar, char newChar)`: Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.

Composite Expressions: Arithmetic Operators

- **Expressions** may be composed through **operators**.
- Java provides five **basic arithmetic operators**:
  - $+$: addition
  - $-$: subtraction
  - $*$: multiplication
  - $/$: division
  - %: modulus (remainder)
- There are also unary $+$ and $-$ operators, i.e. have just one operand.
- The operators can be applied to any of the integer or floating-point types.

Operator Precedence

- **Multiplication, division and modulus** have **higher precedence** than addition and subtraction.
- All higher-precedence operators are evaluated before any lower-precedence operators.
- Operators at the same precedence are evaluated left-to-right.

$$
\begin{aligned}
x + y * z &= x + (y * z) \\
a * b + c\%d &= (a * b) + (c\%d)
\end{aligned}
$$

- **Parathenses** can be used to override operator precedence

$$(x + y) * z$$

- **Assignment** is the **lowest precedence** operator of all.
- **Unary + and −** are **higher-precedence** than multiplication.

- **Integer division**: fractional results round towards zero

$$\begin{aligned} 9/2 &= 4 \\ -9/2 &- -4 \end{aligned}$$

- Integer division and remainder obey the rule

$$(x/y) * y + x\%y = x$$

Composite Expressions – Comparisons

- Comparisons are applied to two expressions of compatible type and always yield a `boolean` result.

```
a < b
a <= b
a == b /* equals */
a > b
a >= b
a != b /* not equal to */
```

- Assignment operator `a=b`

DO NOT CONFUSE WITH `a==b`

Composite Expressions – Comparisons

Examples

```
double a; int i; boolean b;

 a = 3.1415 + 42;  // a = 45.1415
 i = 4 - 9;        // i = -5
 i = i + 1;        // i = -4
 a = i * 2 + 3;    // a = -5
 a = i * (2+3);    // a = -20
 b = i > 0         // b = false
```

Composite Boolean Expressions

- Logical operators enable the composition of single Boolean values
- They are known from last semester:
  - **Logical AND** (A AND B) yields true only if both A and B evaluate to true. In Java: `A && B` or `A & B`.
  - **Logical OR** (A OR B) yields true if either A or B, or both yield true. In Java: `A || B` or `A | B`
  - **Logical XOR** (A XOR B) yields true if and only if exactly one of its operands is true. In Java `A^B`.
  - **Logical Negation** inverts its operand. In Java `!A`
- `A && B` and `A || B`: evaluate the second operand only if required.
- `A & B` and `A | B`: Both operands have to be evaluated.

Composite Boolean Expressions – Examples

```
boolean a, b; int i; char c

c='A'; i = 2; a = false;

 b = c =='A';           // b is now true

 b = a && b;            // b is now false

 b = !b;                // b is now true

 b = i>0 && 3/i == 1    // b is now true
```

Increment and Decrement

- Adding or subtracting one from a number is a common operation.
- Java provides a short-hand in the **increment** (++) and **decrement** (--) operators.
- Increment and decrement can be used as **prefix** or **postfix** operators.

```
a = 4
a++;     // equiv. to a = a + 1;          ---> a = 5
b = a++  // equiv. to b = a; a = a + 1; ---> a = 6, b = 5
b = ++a  // equiv. to a = a + 1; b = a  ---> a = 7, b = 7
b = a--  // equiv. to b = a; a = a - 1  ---> a = 6; b = 7
```

Compound Assignment Operators

- Instead of `i = i + Expression;`
- One may write `i += Expression;`
- **Example:** Instead of `i = i + (2 * j + x);`
  we can write `i += 2 * j + x;`.
- Available as: `+=, -=, *=, /=, |=, &=, ^=, %=`
- Useful if the target is complex or should only be evaluated once.

Bitwise Expressions

To be discussed during the labs

- **Bitwise Operators**
  - `a & b`: bitwise AND
  - `a | b`: bitwise OR
  - `a ^ b`: bitwise XOR, not equal
  - `~a`: bitwise negation
  - `a >> b`: shift `a`, `b` positions to the right
  - `a << b`: shift `a`, `b` positions to the left
- These operators are defined for the following types: **boolean, integers, char**