cisc3120
design and implementation of software applications I
spring 2015
lecture # I.1

instructor email:

- Arif T. Ozgelen, *ozgelen@sci.brooklyn.cuny.edu*

course web page:

- `http://www.sci.brooklyn.cuny.edu/~ozgelen/cisc3120/`

office Hours:

- Room: Roosevelt Hall Rm. 233
- Time (TBA): Mondays 9.30am - 10.30am OR Wednesdays 9.30am - 10.30am
- By Appointment Only!

topics:

- introduction to the course
- introduction to java, part 1

introduction to the course

- about this course
  - intended to give you hands-on experience designing and building a software application
- topics covered:

(I) Object-Oriented Programming (OOP) concepts
(II) Graphical User Interfaces (GUI)
(III) Computer Graphics
(IV) Net-centric Systems
(V) Software Design Concepts

course material

- textbook (RECOMMENDED not Required!): Core Java Volume 1 - Fundamentals 9th Edition by Cay Horstmann and Gary Cornell, Prentice Hallamawww

- Lecture notes (based on notes provided by Prof. Sklar and Prof. Weiss) and other online sources (Check the course website for updates)

course structure

- 3 **units**
- each unit has:
  - **lectures**
  - **labs**
  - **assignment**
- your grade =
  - assignments + projects (55% total)
  - midterm (20%)
  - final (25%)

Java.

- Java is an *object-oriented* language: it is structured around *objects* and *methods*, where a method is an action or something you do with the object
- Java programs are divided into entities called *classes*
- some Java classes are *native*
  but you can also write classes yourself
- Java programs can run as *applications* or *applets*

Java language features I

- Simple : Java omits the rarely used and confusing features of C++.
- Object-oriented : focuses on the data and the interfaces to that object.
- Network-Savvy: extensive library for networking making it easy to cope with protocols.
- Robust : utilizes a pointer model that eliminates the possibility of overwriting or corrupting memory.
- Secure : enables construction of virus-free systems.

Java language features II

- Architecture neutral : compiler generates bytecode which can be interpreted on any machine.
- Portable : there are no implementation dependent aspects; the sizes of primitive data types are specified.
- Interpreted : The Java Virtual Machine (JVM) can interpret bytecodes directly on any machine that it can be ported.
- High Performance : The performance of the interpreted bytecodes are more than adequate. Just-in-time compilers can translate frequently accessed parts of the code into machine code for improved performance.

overview of Java technology

- *JDK Java Development Kit*: The software for programmers who want to write Java programs.
- *JRE Java Runtime Environment* : The software to run Java programs. Made up of JVM and appropriate application programming interfaces (APIs, source code based software specifications).
- *SE Standard Edition* : Java platform for use on desktops and simple server applications.
- *EE Enterprise Edition* : Java platform for complex server applications.
- *ME Micro Edition* : Java platform for cell phones and other small devices.
- *NetBeans* : Sun's integrated development environmet (IDE).

installing Java

- For this class you will need to install Java SE 8.
- Go to the website: http://www.oracle.com/technetwork/java/javase/downloads/index.html and download Java SE 8 Update 31 - JDK
- JDK includes the JRE.
- Follow the installation instructions.
- Make sure to set the `PATH` and `CLASSPATH` environment variables correctly.

compiling and running Java applications

- java programs are *compiled* into bytecodes.

```
$> javac program.java
```

- the output of the above operation is `program.class`.
- the other type of java bytecode is `.jar` files, which we will discuss later.
- to run the program as an application, invoke the *Java Virtual Machine (JVM)* which will interpret the bytecode.

```
$> java program
```


our first application.

"hello world"

- typical first program in any language
- output only (no input)

the application source code.

file name = Hello.java

```
/*------------------------------------------------------------
   Hello.java

   This class demonstrates output from a Java application.
------------------------------------------------------------*/
public class Hello {
    public static void main(String[] args) {
        System.out.println("hello world!");
    }
}
```

output.

- *methods*
  `System.out.println( )`
  `System.out.print( )`
- *arguments*
  - those things inside the parenthesis ( )
  - one or more Strings, separated by "+" 's
  - escape sequences: \n, \t
  - also called *parameters*. more accurately in:

    ```
    void foo(int i) { ... }
    ...
    foo(5)
    ```

    `i` is a *formal parameter*, while '5' is the *actual argument*
- example
  `System.out.println( "The quick" + ", brown " + "fox" );`

things to notice.

- file name is same as class name
- Java is CASE sensitive
- punctuation is really important!
- *whitespace* doesn't matter for compilation
- *BUT* whitespace DOES matter for readability of your code!

data types and storage.

- programs = objects + methods
- objects = data
- data must be *stored*
- all storage is numeric (0's and 1's)

memory.

- think of the computer's memory as a bunch of boxes
- inside each box, there is a number
- you give each box a name
  $\Rightarrow$ defining a *variable*
- example:

program code:

`int x;`

computer's memory:

variables.

- variables have:
  - name
  - type
  - value
- naming rules:
  - names may contain letters and/or numbers
  - but cannot begin with a number
  - names may also contain underscore (_) and dollar sign ($)
  - underscore is used frequently; dollar sign is not too common in Java
  - can be of any length
  - cannot use Java keywords
  - Java is *case-sensitive!!*

variable naming convention

- if variable name consists of a single word → all lowercase. example:
  `double salary;`
- if variable name consists of multiple words → first letter of first word is lowercase, all the first letters of following words uppercase. example:
  `int numEmployees;`
- if variable is a constant → all uppercase, words are divided by an underscore (_) character. example:
  `final int NUM_DAYS_WEEK = 7;`

primitive data types.

- numeric

byte	8 bits	-128 = -2^7	127 = 2^7 - 1
short	16 bits	-32,768 = -2^{15}	32,767 = -2^{15} - 1
int	32 bits	-2^{31}	2^{31} - 1
long	64 bits	-2^{62}	2^{63} - 1
float	32 bits	≈ -3.4E+38, 7 sig dig	≈ 3.4E+38, 7 sig dig
double	64 bits	≈ -1.7E+308, 15 sig dig	≈ 1.7E+308, 15 sig dig

- boolean

boolean	1 bit

- character

char	16 bits

assignment.

- = is the assignment operator
- example:

program code:

```
int x; // declaration
x = 19; // initialization
```

or

```
int x = 19;
```

computer's memory:

x → 19

Strings.

- a `String` in Java is a special data type — it's called a *wrapper class* (which we'll talk about in detail later)
- a `String` is essentially a group of `chars`
- it comes with a *method* called `length()` that lets you find out how many characters are in the string (i.e., how long it is)
- it comes with a number of other methods, which we'll talk about later
- a `char` has single quotes around it

```
char c = 'A';
```

- a `String` has double quotes around it

```
String s = "hello world!";
```

- in this case, the method `s.length()` returns 12

mathematical operators.

$+$	unary plus
$-$	unary minus
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$\%$	modulo

example:

```
int x, y;
x = -5;
y = x * 7;
y = y + 3;
x = x * -2;
y = x / 19;
```

what are `x` and `y` equal to?

modulo means "remainder after integer division"

coercion or type casting.

- remember from last time: data of type `char` is stored as a number — which is really an index into the ASCII table (Unicode to be exact but for the latin alphabet the ASCII values and Unicode values are the same)
- a declaration like this:

```
char y = 'A';
```

really stores a 65 (the ASCII value of 'A') in a memory location that is labeled `y`

- you can do math on that 65 by *coercing* (aka *type casting*) the `char` to an `int`
- for example:

```
char y = 'A';    // initialize variable y to store an A
int  x = (int)y; // initialize variable x to store 65
x = x + 1;       // increment x (to 66)
y = (char)x;     // coerce x from an int to a char ('B')
```


increment and decrement operators.

- increment: $++$

```
i++;
```

is the same as:

```
i = i + 1;
```

- decrement: $--$

```
i--;
```

is the same as:

```
i = i - 1;
```

assignment operators.

+=
`i += 3;` is the same as: `i = i + 3;`

-=
`i -= 3;` is the same as: `i = i - 3;`

*=
`i *= 3;` is the same as: `i = i * 3;`

/=
`i /= 3;` is the same as: `i = i / 3;`

%=
`i %= 3;` is the same as: `i = i % 3;`

boolean expressions.

- boolean variables: `true` (1) or `false` (0)
- logical operators:

!	not
&&	and
||	or

example:

```
boolean x, y;
x = true;
y = false;
System.out.println( "x && y = " + ( x && y ));
System.out.println( "x || y = " + ( x || y ));
System.out.println( "x && !y = " + ( x && !y ));
```


truth tables.

a	!a
false	true
true	false

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

relational operators.

==	equality
!=	inequality
>	greater than
<	less than
>=	greater than or equal to
<=	Less than or equal to

example:

```
int x, y;
x = -5;
y = 7;
```

some truths:

(x < y)	true
(x == y)	false
(x >= y)	false

the if branching statement.

```
if ( x < y ) {
  x = y;
}
```

```
if ( x < y ) {
  x = y;
}
else {
  x = 91;
}
```


the if branching statement (1).

there are four forms:

(1) simple if

```
if ( x < 0 ) {
  System.out.println( "x is negative\n" );
} // end if x < 0
```

(2) if/else

```
if ( x < 0 ) {
  System.out.println( "x is negative\n" );
} // end if x < 0
else {
  System.out.println( "x is not negative\n" );
} // end else x >= 0
```


the if branching statement (2).

(3) if/else if

```
if ( x < 0 ) {
  System.out.println( "x is negative\n" );
} // end if x < 0
else if ( x > 0 ) {
  System.out.println( "x is positive\n" );
} // end if x > 0
else {
  System.out.println( "x is zero\n" );
} // end else x == 0
```


the if branching statement (3).

(4) nested if

you can nest any kind/number of if's

```
if ( x < 0 ) {
  System.out.println( "x is negative\n" );
} // end if x < 0
else {
  if ( x > 0 ) {
    System.out.println( "x is positive\n" );
  } // end if x > 0
  else {
    System.out.println( "x is zero\n" );
  } // end else x == 0
} // end else x >= 0
```

System.exit() (1)

- a *method* in class `java.lang.System`
- definition:

```
public static void exit( int status );
```

- terminates the currently running Java Virtual Machine
- the argument serves as a status code — by convention, a nonzero status code indicates abnormal termination
- use at the end of a program to exit cleanly or to terminate in the middle

System.exit() (2)

```
import java.lang.*;

public class ex_exit {

  public static void main ( String[] args ) {
    if ( args.length < 3 ) {
      System.out.println( "usage: java ex_exit <a> <b> <c>" );
      System.exit( 1 ); // abnormal termination
    }
    // ... rest of program goes here ...
    System.exit( 0 ); // normal termination
  } // end of main()

} // end of class ex_exit
```