cis20.1
design and implementation of software applications I
fall 2007
lecture # I.2

topics:

- introduction to java, part 1

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*

our first application.

"hello world"

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

the application source code.

file name = hello.java

```
/*-------------------------------------------------------
   5-sep-2007/sklar, hello.java

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

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*
- example
  `System.out.println( "The quick" + ", brown " + "fox" );`

things to notice.

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

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!!*

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; // assignment
```

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
- 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 )); // outputs fal
System.out.println( "x || y = " + ( x || y )); // outputs tru
System.out.println( "x && !y = " + ( x && !y )); // outputs t
```


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
```