Welcome to cs3101-003 Java!

Programming Languages: Java

Spring 2003

Wed 11.00am - 1.00pm CS486 (CLIC lab)

Professor Elizabeth Sklar

email: *sklar@cs.columbia.edu*
web: *http://www.cs.columbia.edu/~sklar*
office: 460 Computer Science building (through Mudd)
office hours: posted weekly on my web page

Class web page:
http://www.cs.columbia.edu/~sklar/cs3101

course overview.

- objective
  - become fluent in Java
- resources
  - lectures
  - lecture notes
  - class web page
  - books
  - web
  - TAs
  - me
- requirements
  - you must get a CS account for this class
  - go to http://www.cs.columbia.edu/accounts

assessment.

- 5 homeworks, 15 points each
- 5 in-class exercises/quizzes, 5 points each
- no exams
- class participation counts!

a word about homeworks.

- should be done on your own, as much as possible
- get help from TAs, me, friends
  but you must acknowledge all help received by citing the names of those who helped you in the comments of your code
- this not only protects you from being accused of cheating, but also protects you in case your helper gives you misinformation
- this also lets me know who is really helpful, which is useful in selecting TAs for next semester

homeworks: submission policy.

- homeworks are due on the day that they are due
- here are the rules — please know them well:
  - all homeworks MUST be submitted electronically by 6AM on the due date
  - submission time is clocked according to the time of your electronic submission
  - be aware that the system tends to get clogged when too many people try to submit at the same time — so AVOID A LATE PENALTY and don't submit too close to the 6AM deadline!
  - you may have a total of 25 hours grace time for lateness of electronic copies, which may be used up all at once or split between several assignments
- exceptions and extensions are possible, primarily based on MEDICAL EMERGENCIES — circumstances must be documented and suitable arrangements will be made — you must consult me via email on an individual basis

homeworks: regrade policy.

- if you feel that there was an error in grading your homework, then you need to write on a piece of paper a description of the error and give it to me
- know that the TAs are given a list of expectations for each homework assignment and quiz and told where to take off points — so if your complaint is that too many points were taken off for one kind of mistake or another in your program, then generally those types of things will not change in a regrade
- if there is a genuine error in the marking, like we thought something was missing, but it is really there, then you will likely get points restored
- HOWEVER, a regrade means that the entire assignment or quiz will be remarked, so be aware that your mark can go DOWN as well as UP
- regrades take a while to process, so be patient — if you need the work to study from, then make a copy of it before you turn it in for a regrade

homeworks: a word to the wise.

- save early and save often!
- disk drives crash
- floppies have bad sectors
- power supplies fail
- monitors die
- mice get trapped
- paper print-outs are the best security known to mankind

a word about lectures.

- brief notes for every lecture will be placed on the syllabus section of the class web page
- but they are NOT A SUBSTITUTE FOR COMING TO CLASS
- if you must miss a class, YOU are responsible for getting notes from someone who did come to class
- I will try to post lecture notes on the web before class
- I strongly encourage you to take notes yourself because you learn better when you actually write things down
- everything I say is NOT in the lecture notes
- sometimes there are mistakes in the lecture notes which get caught and corrected during class; I will post updated lecture notes if this happens

a word about academic integrity.

- the work you submit for assessment should be completed ON YOUR OWN
- you may get help from TAs, me, friends
- you must acknowledge all help given
- you should not mail code or copy files
- if someone asks you to do this, *JUST SAY NO!*

topics covered.

1. Java applications; output; data storage and representation; operators; command-line input; branching with `if`
2. branching with `switch`; looping; native classes and methods; classes and objects; inheritance
3. writing your own classes; Java keywords
4. arrays; I/O; exceptions
5. applets; graphics; graphical user interfaces; event handling
6. recursion; data structures; threads

how to learn a programming language.

- YOU are responsible for your own learning!!!
- I will point you in the right direction...
- but YOU must PRACTICE, PRACTICE, PRACTICE...
- and PRACTICE some more!!!
- if you don’t understand, then ASK for help!

which environment?

- there are lots of Java compilers and programming environments
- in class, we'll use Unix and emacs at first
- later we'll look at some free development environments

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

```
/*-------------------------------------------------------
   EISklar, 22-Jan-03, 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:

x $\rightarrow$ 

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	$\approx$ -3.4E+38, 7 sig dig	$\approx$ 3.4E+38, 7 sig dig
double	64 bits	$\approx$ -1.7E+308, 15 sig dig	$\approx$ 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 $\rightarrow$ 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 `char`s
- 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 a, b;
x = 1; // true
y = 0; // false
System.out.println( "x && y is false" );
System.out.println( "x || y is true" );
System.out.println( "x && !y is true" );
```

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
```

flowcharts

- diagram for illustrating control flow of a program
- conventions:
  - rectangle = statement or method call
  - diamond = yes/no or true/false question

command line arguments (1).

- remember our model of a computer program from the 2nd lecture:
  $input \rightarrow \boxed{\text{CPU}} \rightarrow output$
- homework #1 was an *output only* program
- now we will learn how to get *input* into your program
- there are many ways to do this...
- we will start with *command line arguments*, which are a way of getting input into your program from the UNIX environment when you start up your program
- UNIX commands use *arguments* (arguments are also called *parameters*)
- for example, with the command:

  ```
  unix$ ls -l
  ```

  the `ls` part is the *command*; and
  the `-l` part is an *argument* (in this case, `-l` is a special type of argument, also called a "switch" in UNIX; it is an argument that starts with a `-`, and usually is used to indicate switching on or off some feature of the command being run)

command line arguments (2).

- the “hello world” program takes no arguments and is started up like this:

```
unix$ java hello
```

- here’s the source code:

```
public class hello {
    public static void main ( String[] args ) {
        System.out.println( "hello world!\n" );
    } // end of main()
} // end of class hello()
```

command line arguments (3).

- the “hello2” program that takes one argument and is started up like this:

```
unix$ java hello2 ringo
```

- here’s the source code:

```
public class hello2 {
    public static void main ( String[] args ) {
        System.out.println( "hello "+args[0] );
    } // end of main()
} // end of class hello2()
```

- in this example, the argument is `ringo`
- and the output of the program would be:

```
unix$ java hello2 ringo
hello ringo!

unix$
```

command line arguments (4).

- the argument `args` to the `main()` method is of type `String[]`
- which means it is a list of strings (i.e., Java class `String`)
- where a string is a list of characters (i.e., Java primitive data type `char`)
- `String` is something called a *wrapper class*
- we'll talk more about wrapper classes later
- a `String` value is defined using double quotes, e.g.,
  ```
  String x="ABC";
  ```
  or
  ```
  String y="A";
  ```
- a `char` value is defined using single quotes, e.g.,
  ```
  char z='A';
  ```

command line arguments (5).

- when a java program is invoked from the UNIX command line, any values after the program name are *passed into the program*, for use when the program is running.
- the `args` argument to `main` gives you access to these values, for free (i.e., you don't have to do anything special to get them), through the line of code that looks like this:
  `public static void main( String[] args )`
- you can see how many arguments were passed into the program by using the value of `args.length`
- you can see what the values of the arguments are by looking them up in the `args` list, using an *index*, i.e., a number which indicates which entry in the list you are referring to
- remember that in computer science, we start counting with 0
- so the first value in the argument list is referenced as `args[0]`, and so on

command line arguments (6).

- given the command line:
  `unix$ java ex60 A 12 DOG`
  then
  `args.length` would be equal to 3, and
  `args` would look like this:
  ```
  arg[0] "A"
  arg[1] "12"
  arg[2] "DOG"
  ```
- these are all `String`s!!
- if you want to use a command line argument as a number, then you have to convert it, just like we converted, or *coerced*, `int` to `char` and so forth
- for today, only worry about the syntax:

to go from	to	use the following
`String s`	`float f`	`f = (Float.valueOf(s)).floatValue();`
`String s`	`int i`	`i = (Integer.valueOf(s)).intValue();`

`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
```

to do.

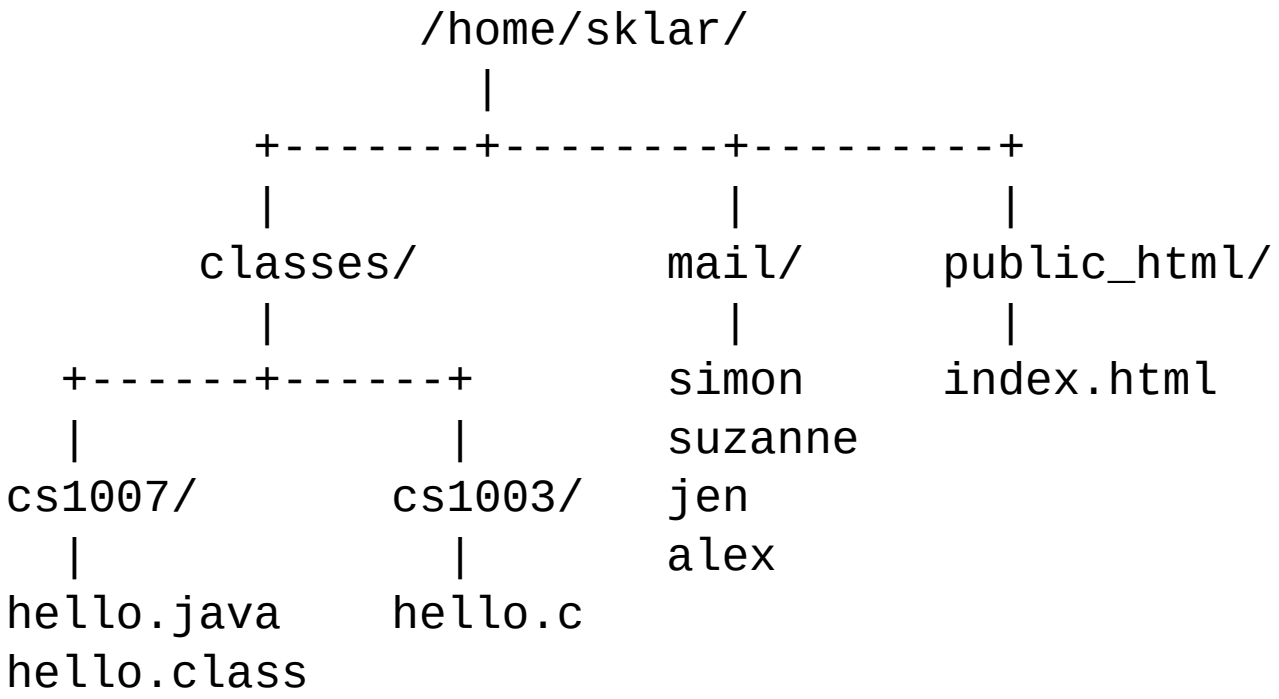
- get a CS account (if you don't already have one)
- ... and try logging in
- check out the class web page:
  http://www.cs.columbia.edu/~sklar/cs3101
- homework #1 will be posted by midnight and will be due next week

try it yourself.

1. log into your CS account
2. create the application source code file “hello.java”,
   using the *emacs* editor type in the code (above)
3. compile the source code,
   using the *javac* command
4. execute the program using the *java* command
5. modify the example, trying different forms of output

files in UNIX.

- hierarchical file system
- example:



quick and dirty UNIX.

- commands have options or parameters or switches
- *switches* start with "–"
- some commands...
  - man
  - pwd
  - cd
  - ls
  - mkdir
  - rmdir
  - cp
  - mv
  - rm
  - chmod

man — get help (display manual page).

man — display manual pages (get help!)
man man — display manual page for the *man* command
man ls — display manual page for the *ls* command
man -k file — list all commands with the keyword *file*

```
unix> man pwd
PWD(1)                          FSF                          PWD(1)

NAME
       pwd - print name of current/working directory

SYNOPSIS
       pwd [OPTION]

DESCRIPTION
       Print  the full filename of the current working directory.
...
```

pwd — print working directory.

```
unix> pwd
/home/sklar/teaching/cs1007/slides
```

cd — change working directory.

```
unix> pwd
/home/sklar/
unix> cd classes
unix> pwd
/home/sklar/classes
```

ls — list the files in the current directory

ls -aF — list all files and show their file types

```
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
```

ls -l — list files in long format

```
unix> ls -l hello.java
-rw-r--r--    1 sklar  faculty   187 Sep 5 10:45 hello.java
```

mkdir — make (create) a directory

```
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
unix> mkdir junk
unix> ls -aF
./
../
.cshrc
classes/
junk/
mail/
hello.java
```

rmdir — remove (delete) a directory.

```
unix> ls -aF
./
../
.cshrc
classes/
junk/
mail/
hello.java
unix> rmdir junk
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
```

cp — copy a file.

```
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
unix> cp hello.java hi.java
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
hi.java
```

mv — move (rename) a file.

```
unix> ls -aF
./
../
.cshrc
classes/
mail/
hello.java
unix> mv hello.java howdy.java
unix> ls -aF
./
../
.cshrc
classes/
mail/
howdy.java
```

rm — remove (delete) a file.

```
unix> ls -aF
./
../
.cshrc
classes/
mail/
hi.java
howdy.java
unix> rm hi.java
unix> ls -aF
./
../
.cshrc
classes/
mail/
howdy.java
```

chmod — change file mode

- 9 characters: `-uuugggooo`
- WHO: u = user, g = group, o = other users, a = all users (u + g + o)
- WHAT: r = read, w = write, x = execute
- MODE: + = allow, - = don't allow

```
unix> ls -l hi.java
-rwxr-xr-x    1 sklar  faculty   187 Sep 5 10:45 hi.java
unix> chmod a+w hi.java
unix> ls -l hi.java
-rwxrwxrwx    1 sklar  faculty   187 Sep 5 10:45 hi.java
```

quick and dirty *emacs*.

```
unix> ejava hello.java
```

or

```
unix> emacs -nw hello.java
```

Ctrl-B	move cursor Back
Ctrl-F	move cursor Forward
Ctrl-P	move cursor to Previous line
Ctrl-N	move cursor to Next line
Ctrl-D	Delete character under cursor
Ctrl-K	Kill (delete) to end of line
Ctrl-Y	Yank back (undelete) killed text
Ctrl-X Ctrl-S	Save the file
Ctrl-X Ctrl-C	eXit emacs
Ctrl-H	Help
Ctrl-G	Gets you out of trouble!