CS 100

"Intro to Computer Programming"

- intro programming – uses Java (some Matlab)
- "programming is easy, persuading the computer to agree is not!"
- think first, then program, expect it to take much time / intensity.
- work in pairs, understand individually.
- for sanity, treat it as a game and have fun.

Java → uses classes and objects
= hyperorganised!

A class Car is like a manufacturer who only constructs individual new cars.

A class Bucket will only construct individual new buckets.

Cars and buckets have natural things which belong to every car or bucket, although of course cars have different colours and buckets different sizes!

To build a red car called ferrari, we might write

```
Car ferrari = new Car(red);
```

— I'm not promising that this will work!!!

and to build a yellow car called ccbb, we might write

```
Car ccbb = new Car(yellow);
```

Of course,

```
Bucket rollsroyce = new Bucket(huge);
```

will only give you a peculiarly named huge bucket.

If you want to access stuff in your car, then the "dot" . is the "genitive case", so

```
ferrari.colour
```

would be red, and

```
ccbb.colour
```

would be yellow. This is also like a path; looking into the ferrari or the ccbb to find the individual colours.

More on this later.....

Enough of such generalities! How do we write a simple program in Java?

First we need to be able to get stuff into and out from the computer!

```
System.out.println ("Once upon a time ...");
```

looks into the System where it finds an out, and looks into System's out where it finds a method (or function or routine) which can print a String of characters onto a fresh line on the standard output screen.

```
System.out.print ("Golly gosh");
```

does exactly the same, except the method print doesn't finish with a "new line".

To read in a String of characters from the standard input keyboard:

```
InputStreamReader nab = new InputStreamReader(System.in);
BufferedReader grab = new BufferedReader(nab);
```

constructs a BufferedReader called grab so that

```
grab.readLine( );
```

reads a whole line of input.

Java is a language of "let's pretend!", so grab is a virtual keyboard which has the ability (among other skills) of readLine() — all the other stuff is there to establish a connection between "make believe" and "reality".

We can do the same thing with files:

```
FileReader secret = new FileReader("spy.oops");
BufferedReader james = new BufferedReader(secret);
```

constructs a BufferedReader called james so that

```
james.readLine();
```

reads a whole line of input from spy.oops. As a matter of common courtesy, you should

```
james.close();
```

close the 'file' when you've finished with it!

(If you need to specify a path for your file, you can have

```
= new FileReader("c:/money/penny/spy");
```

or whatever is appropriate for your system.)

Similarly,

```
FileOutputStream plop = new FileOutputStream ("meow.t");
PrintWriter scribble = new PrintWriter (plop);
```

allows

```
scribble.println ("What big teeth you have!");
```

to write to the file meow.t, which again should be closed by

```
plop.close ();
```

when finished with.

Of course, we could have done the same thing when writing to the screen:

```
PrintWriter tube = new PrintWriter (System.out, true);
tube.println ("How time flies!");
```

Here, tube is the name of the make-believe computer screen.

Now that we can get stuff into and out of the computer, let's actually write a program.

changed

```
import java.io.*; // so that I/O stuff is available

public class PlayTime
  {
    public static void main (String [] args) throws Exception
      {
        InputStreamReader  ca = new InputStreamReader (System.in);
        BufferedReader  va = new BufferedReader (ca);
        PrintWriter  bon = new PrintWriter (System.out, true);

        int  x, y = 2;

        bon.println ("Enter an integer.");
        x = Integer.parseInt (va.readLine());
        y = y * x - x / 2;

        bon.println ("x was " + x + " and y is " + y);
        ca.close();
      }
  } // end of class PlayTime
```

action stuffs HERE

in case of silliness

This whole file would be called

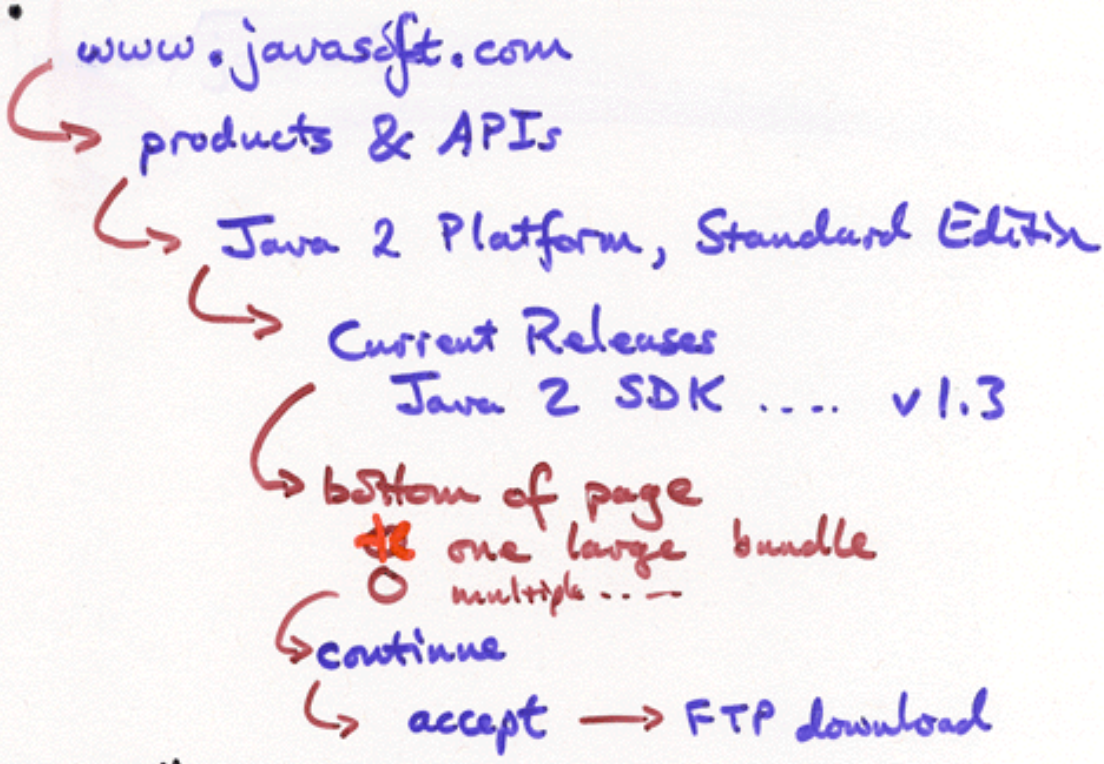
PlayTime.java

and compiled and run as relevant to your computer system.

Now for some routine details . . .

There are development environments, like CodeWarrior available on the lab computers — these are commercial programs which provide an integrated development and debugging environment.

Alternatively, you can get Java for free as follows...



then follow installation instructions. After installation, make sure that both

PATH and CLASSPATH

environment variables are set to find jdk1.3 and the directories/folders you'll be running JAVA from.

Finally, save an easy Java program as

test.java

and in a command prompt or shell window, do

javac test.java then java test.

Primitive Types

byte	$-128 \leq$ integer < 127
short	$-32768 \leq$ " < 32767
int	$-2^{31} \leq$ " $< 2^{31}-1$
long	$-2^{63} \leq$ " $< 2^{63}-1$
float	$\pm\ 10^{-46} <$ decimal $< 10^{38}$
double	$\pm\ 10^{-324} <$ " $< 10^{308}$
char	single unicode character
boolean	false, true

The declaration

```
int boo;
```

makes boo an allowable name for an integer. The declaration and initialization

```
int boo = 13074;
```

be careful that = is really assignment, not 'equals'

makes boo an allowable name for an integer, and before it can be used, initializes its value to 13074.

Similarly, we can have

```
double whoosh = 9.874;
char cuckoo = 'T';
boolean ouch = false;
```

note the single quotes

Awkward characters like ? or ' can be assigned using \ as in

```
cuckoo = '\?';
cuckoo = '\'';
```

Changed

Arithmetic

+	plus	3+4; → 7
−	minus	3−4; → −1
*	times	3*4; → 12
/	divide	3/4; → 0
%	remainder	3%4; → 3

What do you think (-3)%4; evaluates to?

[As an aside, it's worth noting that there is a non-primitive type String which carries a string of characters, and

```
String tut = "Methinks I were, ";
String um = " there is no man...";
tut = tut + um;
```

gives tut the updated value of

Methinks I were, there is no man...

] So for strings, + appends the second string to the end of the first string.

Also, for those who like calculating...

```
Math.sin(1.78);
Math.atan(72.4);   etc
```

all do the obvious — Math is a repository of lots of useful stuff!

Now for an example . . .

```
import java.io.*;

public class Multiplier
  {
    public static void main (String [] args) throws Exception
    {
      InputStreamReader isr = new InputStreamReader (System.in);
      BufferedReader comingIn = new BufferedReader (isr);
      PrintWriter goingOut = new PrintWriter (System.out, true);

      int x, y;
      long z = 0;
      String ask = "Please enter an integer.";

      goingOut.println (ask);
      x = Integer.parseInt (comingIn.readLine ());

      goingOut.println (ask);
      y = Integer.parseInt (comingIn.readLine ());

      z = x * y;

      goingOut.print ("The value of " + x + " times " + y);
      goingOut.println (" is " + z);

      goingOut.println ("Thanks for using \"Multiplier\");
      goingOut.println ("Do come again!");
      comingIn.close ();
    }
  }
```

setting up the I/O

space to store input

bigger space for answer!

get the first number

if what was 'read' can't be an integer, throw Except

get second number

actually perform the multiplication!!

give out the answer

unremitting courtesy!

There are also various 'shorthands' ...

```
int a, b;
a = 17;
```

`a = a + 12;`	`a += 12;`	→ a ↦ 29
`a = a - 4;`	`a -= 4;`	→ a ↦ 25
`a = a * 3;`	`a *= 3;`	→ a ↦ 75
`a = a / 5;`	`a /= 5;`	→ a ↦ 15

`a++`	`a=a+1`
`a--`	`a=a-1`

`b = 2 * (a++);` → (a ↦ 16, b ↦ 30)

`a = 4 * (++b);` → (a ↦ 124, b ↦ 31)

`a--;` → a ↦ 123

`--b;` → b ↦ 30

Type conversion is also very handy ...

```
int a = 73, b = 10;
double c;
c = a / b;
```
→ c ↦ 7

```
c = (double) a / b;
```
→ c ↦ 7.3

precedence

↳ anon double variable

Comparisons

`a == b`	`a < b`	`a > b`
`a != b`	`a <= b`	`a >= b`

These have the obvious meanings for the primitive types.

Logic

`a && b`	AND	`a & b`
`a || b`	OR	`a | b`
`!a`	NOT	

So for example, `a != b` and `!(a == b)` are equivalent.

The difference between `&` and `&&` (similarly for `|` and `||`) relies on "short-circuiting".

```
(3 == 7) && (2 == 3/0)
```

evaluates comfortably to false, since failure occured in the first term there was no need to go further; but

```
(3 == 7) & (2 == 3/0)
```

is a disaster, since the single ampersand has no short-circuit provision.

Control

```
if (cccc)
  {cccccccc}
```

Branching processes

and

```
if (ccccc)
  {cccccccc}
else
  {cccccc}
```

We can now broaden our previous example ...

```
import java.io.*;
public class Arithmetic
  {
    public static void main (String [] args) throws Exception
     {
       InputStreamReader isr = new InputStreamReader (System.in);
       BufferedReader comingIn = new BufferedReader (isr);
       PrintWriter goingOut = new PrintWriter (System.out, true);
        char op;
        int x, y;
        double answer;
        String ask = "Please enter an ";
        goingOut.println (ask + "integer.");
        x = Integer.parseInt (comingIn.readLine());
        goingOut.println (ask + "operator.");
        op = (comingIn.readLine()).charAt(0);
        goingOut.println (ask + "integer.");
        y = Integer.parseInt (comingIn.readLine());
        if (op == '+')
            answer = x + y;
        else if (op == '-')
            answer = x - y;
        else if (op == '*')
            answer = x * y;
        else if (op == '/')
            if (y != 0) answer = x/y;
            else answer = x/y;
        goingOut.println ("The answer is " + answer);
        comingIn.close();
     }
  }
```

we should handle this a bit better

this will create an exception

```
while (xxxxx)
   { xxxxxxx }
```

Looping processes

and

```
do
   { xxxxxxx }
while (xxxxxx);
```

and

```
for (xxxx; xxxx; xxx)
   { xxxxxxxx }
```

```
xxxxxxx ? xxxxxx : xxxx;
```

and

More branchy processes

```
switch (name)
{
    case value : xxxxx break;
    case value : xxxxx break;
    case value : xxxxx break;
    default : xxxxx
}
```

Delegation

There are times when it's sensible to delegate certain tasks within a program. Javaspeak for this is **methods** (alias functions or subroutines). For example...

```
public class Stats
{
   public static void main (String[] args)
   {
      int a = 7, b = 12, c = 24;
      double d1;

      d1 = avg (a, b);

      System.out.println ("The average of " + a
         + " and " + b + " is " + d1 + ".");

      System.out.println ("Including " + c
         + " changes this to " + avg(a,b,c) + ".");
   }

   public static double avg (int x, int y)
   {
      return (x+y)/2.0;
   }
   public static double avg (int x, int y, int z)
   { return (x + y + z)/3.0; }
}
```

values copied

unambiguous

overloading