Java in a (pea)nutshell

Peter Boyle

September 22, 2009

1 Java in a (pea)nutshell

These notes are intended as a compressed summary of the Java required to complete Computional Methods. This course is intended to follow on from SciProg 2A. However, it is important to recognise that in the real world memories are fallible, and some assistance with refreshing the memory helps.

1.1 Ask questions

Please ask questions. Any questions. At least about computers and programming! I will probably know the answer and enjoy answering. Don't be shy.

2 How does it all work

2.1 Semicolons and comments

Every java statement ends in a semicolon

Comments are started with a //
The remainder of the line is a comment.

The language is free format - one Java statement can be spread across several lines to make it pretty. The semicolon ends the statement, not a newline.

2.2 Nouns and verbs

An element of data is a "place to put" numbers that can hold values and can in principle be changed. This could be an entry in a file in an evil government database, or as innocent as the "address" within your PC's memory – i.e. the memory location number/index.

This should be familiar, just like saying cell C2 in a spreadsheet.

Computing languages consist of was of "naming" bits of data with meaningful names – this is akin to deciding what are the set of "nouns" in your program.

Languages also provide ways of instructing the computer to modifying bits of data – generally these statements have an "=" sign and are the verbs of your programme.

2.3 The Basic datatypes

The simplest nouns you can declare are the basic, or built-in, datatypes. For the purpose of this course these are:

- int – an integer number

- double – a real valued (floating point/fractional) number

Every Java variable/noun is announced in a declaration. This specifies the type of a variable and its name and ends in a semicolon. e.g.

```
int x;
double force;
double acceleration;
double position;
double mass;
int this_is_a_pointless_counter;
```

Every statement or declaration in Java ends in a semicolon.

2.4 Statements

These are the verbs of Java.

These contain equals signs. The left value which is modified and names a single variable. The right value can be a mathematical expression. It ends in a semicolon.

```
acceleration = force/ mass;
```

2.5 Sequences of statements

A programme consists of sequences of statements. These are performed one after the other unconditionally.

```
int n;
n=0;
n=n+1;
n=n+1;
n=n+1;
```

The above sequence declares a Java integer called "n"
It assigns n with the value zero
One is added to n three times
Therefore at the end of this sequence the memory location representing "n" will contain the value "3".

2.6 Processors

It is worth giving a concrete model to think about what a processor is and what it does. Different processors will differ in the details, but this is a good working model to understand what is going on.

Processors are little machines (technically finite state machines). They possess an interface, or bus, to memory chips.

Both processor instructions and the data on which they operate are held in memory.

We might for now, assume that each processor instruction is a 32 bit number.

Each memory address is a 32bit number represeting a byte address.

The processor has 8 temporary 32 bit "slots", called registers. These are numbered between 0 and 7, and each can hold an integer, or equivalently, a memory address.

The processor keeps track of an "instruction pointer" (IP) which is the address of the next instruction to execute.

To execute an instruction, the processor

1 Loads the 32 bits instruction I contained in the memory address IP.

2 Deciphers from the value of I what the heck it was supposed to do (decode)

3 Executes the instruction
e.g. load reg0 with memory cell whose address is in reg1
e.g. add reg0 to reg1 and stick result in reg2 e.g. change the instruction pointer IP to a new address (branch/function call)

4 Advance IP to next instruction (unless branch)

5 Goto 1

2.7 Compilers

Lets revisit our sequence:

```
int n;
n=0;
n=n+1;
n=n+1;
n=n+1;
```

This is interpreted by a compiler to produce processor instructions. What it says is:

- Reserve me a memory location; I frankly don't care what number it is because numbers are confusing – I want to call it "n" please.
- Store "zero" into the location of "n".
- Add one to the contents of n
- Add one to the contents of n
- Add one to the contents of n

The compiler will do the job of translating all this text into a sequence of 32bit instruction numbers that make all this happen.

3 Branches

All languages introduce the concept of control flow.

These are ways to make the processor *change* it's instruction address (i.e. jump around) beyond it's default behaviour of just moving to them one after the other.

Without this programming would get very tedious and inflexible!

3.1 Ifs and Elses

The simplest control flow is the if-else construct.

The choice is controlled by some form of comparison or "boolean" logical condition.

For example:

```
int absolute_value;
if ( i < 0 ) {
   absolute_value = -i;
} else P
   absolute_value = i;
}
```

The most common comparisons are `==, <, >, >=, <=`

Also common is the *not* operator `!` .

e.g. the following is equivalent

```
int absolute_value;
if ( !(i > 0) ) {
   absolute_value = -i;
} else P
   absolute_value = i;
}
```

3.2 For loops

A better way to programme the above sequence would be:

```
int n;
n=0;
for(int i=0;i<3;i++) {
  n=n+1;
}
```

This could now easily be enhanced to add all the way up to 10,000 painlessly, while the earlier approach could not.

The *for* loop above is quite flexible and can be used in a number of ways. The above is the most common.

The *for* loop contains four key components:

`for( INITIAL ; CHECK ; INCREMENT ) {  BLOCK }`

Here

INITIAL is performed ONCE when first reaching the loop

CHECK is a logical condition that decides if the loop is run

BLOCK is a sequence of statements enclosed in braces that are executed each time the loop is run

INCREMENT is a change to some *loop iterator* performed after BLOCK is executed

Note, `i++;` means `i=i+1`

Other forms are possible:

```
for( int i=2 ; i< 1000 ; i=i*2 ) {  BLOCK } // powers of two
for( ; ; ) {  BLOCK }  // loop forever
```

3.2.1 Arrays & Matrices

Arrays of basic data types can be declared

```
double [] this_is_a_vector = new double [3];
double [][] two_dim_potential = new double [256][256];
double [][] matrix_A = new double [3][3];
double [][] matrix_B = new double [3][3];
```

These are referenced using square brackets, and an integer index. For example, a classic matrix multiply is:

```
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   matrix_C[i][j] = 0.0;
 }
}
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
  for(int k=0;k<3;k++){
    matrix_C[i][j] = matrix_C[i][j] + matrix_A[i][k] * matrix_B[i][k];
  }
 }
}
```

3.3 Method calls

Now we're getting dangerously close to having to introduce classes and real Java programs!

Java allows defining "methods". These are a sequence of statements that are given a name.

A method can be called from another method.

They should be thought of as generic functions of N-parameters, just like in mathematics.

3.3.1 Declaring methods

Each method must return something, and this is a datatype specified in its declaration. The method may or may not take parameters – you the programmer choose by how you write the method.

For example:

```
int thisUsesNoParameters() { return 0; }
int thisUsesOneIntParameter(int x) { return 1; }
int thisUsesTwoIntParameters(int x, int y) { return 2; }
double thisUsesOneIntOneDoubleParameters(int x, double y){ return x*y; }
```

And when I said, every method must return something, that means even when it doesn't you have to say "void":

```
void thisUsesNoParametersButReturnsNothing() { return ;}
void thisUsesOneIntParameterButReturnsNothing(int x){ return ;}
void thisUsesTwoIntParametersButReturnsNothing(int x, int y){ return ;}
void thisUsesOneIntOneDoubleParametersButReturnsNothing(int x, double y){ return ;}
```

So: you have to say void if you return nothing, but say nothing if you pass nothing. I didn't write the language, I only teach it.

Of course , each of these

3.3.2 Calling methods

We can of course then call these methods For example:

```
int a=2;
double d=3.1415926;
double r;

r= thisUsesOneIntOneDoubleParameters(a,d);
```

Will set $r = 2\pi$.
The programme will jump to the method thisUsesOneIntOneDoubleParameters.
Inside this method x will take the value 2, y will take the value 3.1415926;
These will be multiplied together and the product, 2π, assigned to r.

3.4 Where should I start?

If a programme contains a method called chicken() and a method called egg(), which gets called first?

We cannot possibly organise all this without the answer to the question "Where should we begin?"

Java solves this by reserving a special name *main* for the method it automatically calls first. Thereafter it calls whatever *main* tells it to.

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

The *public* and *static* keywords mean the method can be called by any class without complication. If you don't know what you are doing it is a good idea to use these. I will return to this later with further explanation.

The *String[] args* is a parameter that main must take.

For now, just accept this as a rule for how to declare a main method.

4 Basic Classes

Java is an object orientated language that owes much to C++.

In Java, all code and variables are organised in bundles called *classes*.

It is not possible to have Java code that is not organised in a class.

This is the reason it has taken so long to a real, runnable programme. There is significant "magic" associated with putting small code sequences in a class.

4.1 Hello World

This is the simplest Java program

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

This declares a class called HelloWorld. Everything between the first open brace { and the last close brace } *lives* in the class HelloWorld.

The only thing that lives in this class is the *method* main.

The line *System.out.println* is important.

This is the first time we've seen something come *back* from your program. You can print out values of your variables using:

```
    double x=3.2;
    System.out.println("x is "+x); // Display the variable x
```

This is the principle tool you have to debug your programs!

4.2 Edit, Compile, Run

I recommend editing your program code using *xemacs*

Each file should contain one and only one class.

If the class is called *HelloWorld*, the java file should be called *HelloWorld.java*. The java compiler assumes this.

Compile it by typing `javac HelloWorld.java`

Run it by typing `java HelloWorld`

5 Instances

Once we start associating data in an object the concept of an *instance* makes sense.

Later in the course we will come across a Complex object.

5.1 Instance data

This contains real and imaginary elements of data:

```
class Complex {
  double re;
  double im;
};
```

This introduces an object A with *instance* data.

You should read the above as

- each Complex has a real
- each Complex has a imaginary

You might think that would be enough to do the following:

```
  Complex A;// A is an instance of object
  A.re = 0; // Refers to re of instance A
  A.im = 1; // Refers to im of instance A
```

However, Java treats classes in a fashion that is fundamentally different to the basic data types (int, double).

Objects variables are in fact references to objects.

Oh, yeah. What does that mean?

It means: that the variable A is a reference to memory (i.e. a box number) for the instance data (re,im).

Oh, yeah. What does that mean?

It means: A.re = 0 will *not* work until you have requested a new box for A using the *new* construct.

So, you have to do:

```
class Complex {
  double re;
  double im;

  Complex(double x, double y){// Explicit Constructor
    this.re = x;
    this.im = y;
  };
  Complex(Complex copyme){ // Copy Constructor
    this.re = copy.re;
    this.im = copy.im;
  };
};
```

Complex is now a *new* data type that the programmer has defined. That is you can now have variables of type complex, just as previously you could have variables of the built-in data types.

For example

```
  Complex A = new Complex(0,1); // A = i
  Complex B = new Complex(0,1); // A = i
```

Now there are two methods here that are special. They have the same name as the class and are known as constructors.

Once they are allocated you can refer to the instance data through the instance:

```
  A.re =  0;
  A.im =  1;
```

That's why it's called instance data – it is specific to the instance (i.e. variable) and you refer to it through this instance name.

5.2 Instance methods and *this* instance

In the above constructors there we slipped in the use of the *this* key word.

```
  Complex(Complex copyme){ // Copy Constructor
    this.re = copy.re;
    this.im = copy.im;
  };
```

When an instance method of an object is called it *knows* upon which instance it is operating. When the method is called the magic word *this* refers to the copy of the object the method was called for.

For the constructor *this* is the *new* one. For other instance methods the association of *this* is made differently.

Lets add instance some methods:

```
class Complex {
  double re;
  double im;

  Complex(double x, double y){// Explicit Constructor
    this.re = x;
    this.im = y;
  };
  Complex(Complex copyme){ // Copy Constructor
    this.re = copy.re;
    this.im = copy.im;
  };

  double NormSquared(){
   return this.re*this.re + this.im*this.im;
  };
  void Conjugate(){
   this.im = - this.im;
   return;
  }
};
```

These could be called as:

```
  Complex A = new Complex(0,1);
  double nsq = A.NormSquared();
  A.Conjugate();
```

Note, the instance method is called through instance name. This how you tell the Java which variable *this* should refer to. That's why it's called an instance method.

5.3 toString method

The toString method is also special in Java. This is an example for Complex:

```
public String toString()
{
    return "("+re+","+im+")";
}
```

You can then print your class using System.out.println as follows:

```
Complex A = new Complex(1,1);
System.out.println(''A is ''+A);
```

String and System are in fact built-in Java classes. More on that later.

5.4 Consequences of Javas *reference* model

The reference model is one of the biggest pains in dealing with Java.

To keep things simple, I will just introduce three rules:

- Always create a class variable with *new* `Complex A = new Complex(0,0);`
- Dont ever assign two objects as `A=B;`
- Do use `A = new Complex(B);`

Those who revel in the gory details of these things can ask me. But likely they will not need to ask me!

Arrays of objects are also difficult. Not only do you need to allocate the array, but you also need to allocate each element of the array:

```
Complex [][] cplx_2d = new Complex [16][16];
for (int i=0;i<16;i++){
 for (int j=0;j<16;j++){
   cplx_2d[i][j] = new Complex(0.0,0.0);
 }
}
```

6 Static data and methods

Not all data defined in a class need be instance data.

Here the keyword static, that we met earlier and glossed over comes in.

6.1 static data

data members of a class can be declared static

```
class Electron {

  double x;
  double y;
  double z;
  static double mass;
  static double charge;

};
```

The keyword static is a historical accident. *shared* would have been a better choice.

The above declarations should read:

- each Electron has a x
- each Electron has a y
- each Electron has a z
- all Electrons share a mass
- all Electrons share a charge

Static data is accessed through the class name, and not the instance name. A programmer need not have access to an instance of the class to access static data members

```
Electron A = new Electron();
A.x = 0;
A.y = 1;
A.z = 2;
Electron.mass   = 9.1E-31;
Electron.charge = 1.6E-19;
```

6.2 static methods

If a method does not operate on any data of the *this* instance, then it is not necessary to associate the method with an instance when calling.

For example,

```
class Electron {

  double x;
  double y;
  double z;
  static double mass;
  static double charge;
  static double getEonM() { return charge/mass; };
};
```

The method getEonM can be called as:

```
double eonm;
eonm = Electron.getEonM();
```

7 The Java library & documentation

There are many many predefined classes in the Java library.

It's beyond the scope of this, or likely any, course in Java to cover the library in detail. It's impossibly big! My aim here is to teach you the grammar, and how to use the dictionary and expect you to improve your vocabulary on an annual basis. You never stop learning more.

The Java basic libraries are documented at:

http://java.sun.com/javase/6/docs/api/

7.1 Math library

We'll pick the Math class as an important example.

If you know what you are looking for typing something like Math.sin into google works pretty well. But if you don't the Java doc web page is the way to find out.

Click the web link

http://java.sun.com/javase/6/docs/api/

Scroll down the bottom left panel until you see the Math class.

Click on it.

In the main frame you should see documentation for the Math class.

Reading this documentation is a learned skill.

Don't Panic!

7.1.1 Field summary

This means the data members. Math contains two *static* members; not unreasonably "E" and "Pi".

These are static (shared) so you don't need an instance of Math to use them. You just access them through the class name as Math.Pi – for example:

```
double circumference;
double radius;
radius = 1.0;
circumference = 2.0 * Math.Pi * radius;
```

7.1.2 Method summary

There's lots of useful methods such as sin, cos, tan, log, pow documented here. Read some of them – it should be reasonably understandable.

Because these are all static methods, you don't need an instance of Math to call them. You call them as:

```
double theta;
double sin_theta;
theta = Math.Pi/3;
sin_theta = Math.sin(theta);
```

7.2 Other classes

Of course, there's lots of other classes in Java. Far to many to go through. You'll probably just pick up vocabulary piecemeal, like bits of System, FileWriter etc...

Good luck!