PHYS 488

Modelling Physical Phenomena

Lecture 3

Phys488: What we learned in week 2: *if* statement and *for* loop

The *if* statement is useful to make choices within a program depending on some condition.

```
if ( some test )
{
    java code executed if the test is TRUE
}
else
{
    java code executed if the test is FALSE
}
```

The second '*else*' block is optional.

The *for* loop allows you to repeat a command a given number of times

Initialising the counter

condition to continue

Increments the counter

```
for (int bins =0; bins <= SIZE-1; bins++) //
    {
        screen.println("Bin " + bins + " contents =  " +  hist1[bins]);
    }
```

Phys488: What we learned in week 2: *arrays* and the *cast* statement

Arrays are useful for dealing with series of values, for example the bins of a histogram.

```
final int SIZE = 20;
int [] hist1 = new int[SIZE];
```

A ***cast statement*** is sometimes needed to change from one type of variable to another.

```
bin = (int)((nextone-binlow)/binsize));
```

changes the floating point result to an integer.

Phys488: A few rules on good programming practice

It is a considerable help to **indent** each new block in a Java class and pair up the { } in vertical lines. This helps to find misplaced, missing or extra brackets, a common, and hard-to-spot, fault.

```
class SomeClass
{
    int output = 0;
    int SomeMethod (int Input)
    {
        if (some test)
        {
            output = 1;
        }
        return output;
    }
}
```

In general it's not a good idea to built specific numbers into your code ("***hard-coding***"), instead always use variables to store them.

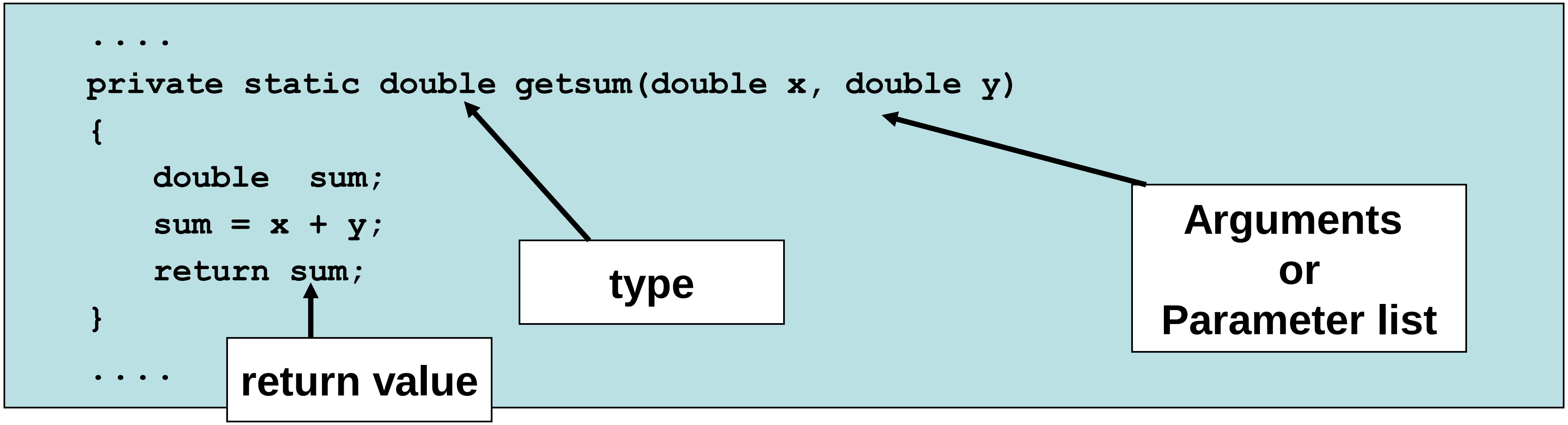
So **avoid** things like:

```
    if (nextone>0.4 && nextone<0.9)
```

Better to use:

```
    double binlow=0.4;
    double binhigh=0.9;
    ..
    if(nextone>binlow && nextone<binhigh)
```

Phys488: What we learned in week 2: methods



```
....
private static double getsum(double x, double y)
{
    double  sum;
    sum = x + y;
    return sum;
}
....
```

type

Arguments
or
Parameter list

return value

The structure of a java class consists of a number of **independent** *methods* that perform specific tasks.

Data can be passed to a method via a *parameter list*, and the method can return a single value via its *return* statement. The parameters MUST be given in the correct order. (methods that don't return a value are declared with the return type *void.)*

Phys488: The *scope* of variables (*local scope*)

```
private static double getsum(double x, double y)
{
    double  sum;
    sum = x + y;
    return sum;
}
....
public static void main (String [] args ) throws IOException
{
    double first = 1.5;
    double second = 2.5;
    double ans = getsum( first, second);
    screen.println(" The sum of these two numbers = " + ans );
}
```

“x”, “y” and “sum” are only accessible to method getsum

“first”, “second” and “ans” are only accessible to method main

The variables defined **within** a method are not accessible to the rest of the program (they have ***local scope***).

In the parameter list of a method only the values are passed, hence a variable cannot be overwritten by the method.

This ensures input parameters (e.g. the variables “first” and “second”) **cannot be modified** either by accident or by design from **inside** the method. This is an important safety feature that helps to prevent mistakes.

(By default any variable declared with a set of curly brackets has (local) scope, only within those brackets)

Phys488: The *scope* of variables (*class scope*)

Variables can be given ***class scope*** by defining them at the start of the class using the keyword ***static***, before the first method. Such variables can **be accessed and/or modified** by any part of the class.

Hence such variables can be used within methods without having to pass them in the parameter list.

```
....
class GenerateHistogram
{
   static PrintWriter screen = new PrintWriter( System.out, true);
   static final double c=3E8;
   public static void main (String [] args )
   {
      screen.println( "The value of c is " + c);
      .....
```

Classes (focus of exercises in the following weeks)

The more radical step to making code more **modular** (and which defines **Object Oriented Programming**) is the use of additional Classes. (Remember the program itself is a Class already).

A Class can contain multiple methods and variable declarations.

Hence a Class can be used (for example from your main method) to:

- Provide a set of tools
- access multiple variables

How and when to use Classes?

- To provide tools you would re-use in different programs
- To cluster variables and tools that naturally fit together

This needs some examples!

Example: A particle class

A good example of a bundle of variables and tools that naturally fit together are the properties of a particle and the various kinematic calculations associated with these.

Variables

A particle has a momentum in x,y, and z, energy, mass, charge, ...

Tools

- calculate the mass based on the energy and momentum
- Lorentz-boost the momentum to another frame
- combine the momenta of 2 particles in a single one (for example) to identify a particle based on observing its decay products

A **particle class** would allow you to do things like:

```
Particle Electron1 = new Particle(momentum, type, etc..)
Particle Electron2 = new Particle(momentum, type, etc..)
Particle Electron3 = Electron1 + Electron3
Particle Electron3Boosted = Electron3.BoostParticle(Px,Py,Pz)
screen.println( "The boosted momentum in x is " + Electron3Boosted.Px());
.........
```

Example of using a class

Different ways to write the HelloWorld program.

Simplest possible

```
import java.io.*;
class HelloWorld1
{
     public static void main(String[] args)
     {
          System.out.println(" Hello World!");
     }
}
```

The preferred way in Object-Oriented style (this is what we used in week 1).

```
import java.io.*;
class HelloWorld
{
     PrintWriter screen = new PrintWriter(System.out,true);
     public static void main(String[] args)
     {
          screen.println(" Hello World!");
     }
}
```

We define an instance “screen” of the class “PrintWriter” allowing us to access various useful tools from this class. The second way is more flexible. For example it would be relatively easy to change the code to write to a file instead of to the screen…. PTO

Two ways to write the HelloWorld program. (cont.)

Writes to screen

```
import java.io.*;
class HelloWorldtoFile
{
     public static void main(String[] args)
     {
                 PrintWriter myPreferedOutput = new
PrintWriter(System.out,true);
          myPreferedOutput.println(" Hello World!");
     }
```

Writes to file

```
import java.io.*;
class HelloWorldtoFile
{
     public static void main(String[] args) throws FileNotFoundException
     {
          PrintWriter myPreferedOutput = new PrintWriter("MyFile.txt");
          myPreferedOutput.println(" Hello World!");
          myPreferedOutput.close();
     }
}
```

Even for very long programs (with many “println” statements) we only have to change a few lines!

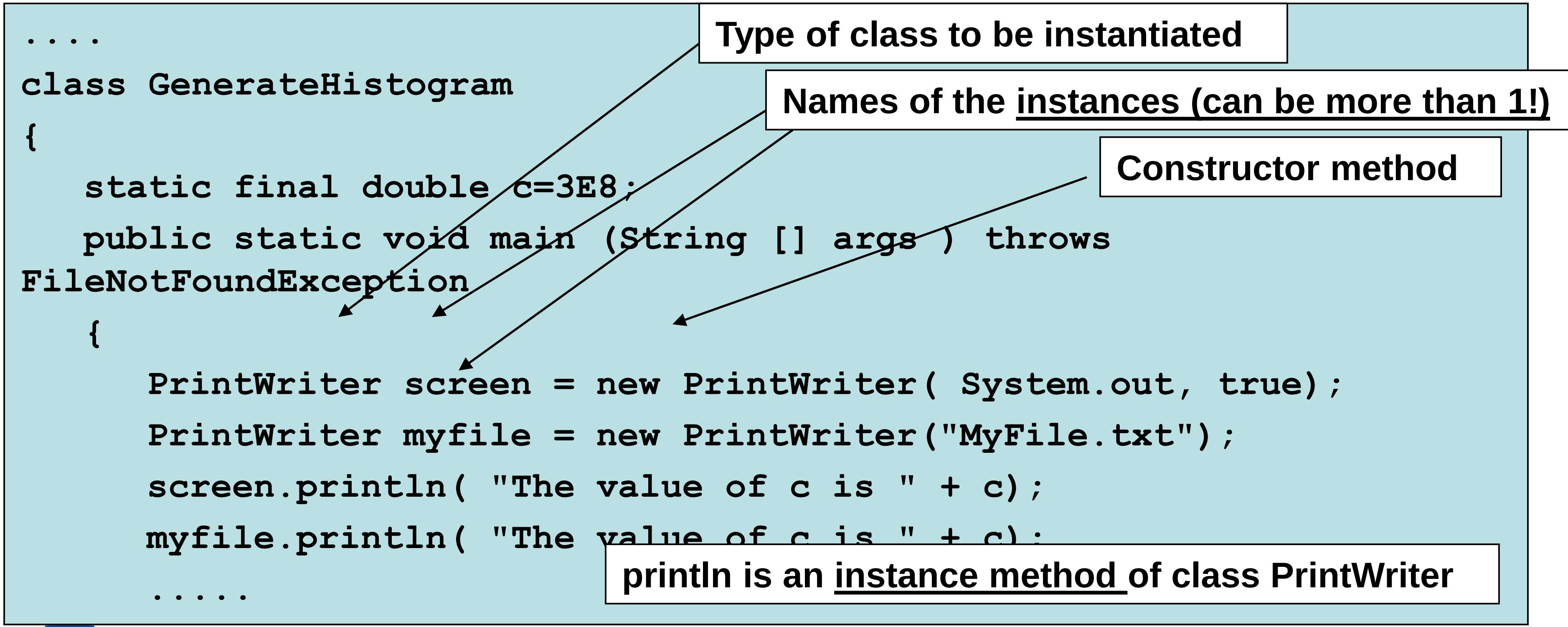
Phys488: Instantiation & instance methods

This week we will **get to the heart of OO programming**, which is to understand the concepts of an *object*, *instantiating* an object and using *instance methods* within that object.

Access to an object is via a *reference* variable which **points to where the object is stored in the computer's memory.**

In fact we've encountered some instance variables already:

"screen" is a reference variable pointing to an object in memory which is an instance of the class PrintWriter



```
....
class GenerateHistogram
{
   static final double c=3E8;
   public static void main (String [] args ) throws
FileNotFoundException
   {
      PrintWriter screen = new PrintWriter( System.out, true);
      PrintWriter myfile = new PrintWriter("MyFile.txt");
      screen.println( "The value of c is " + c);
      myfile.println( "The value of c is " + c);
      .....
```

Type of class to be instantiated

Names of the instances (can be more than 1!)

Constructor method

println is an instance method of class PrintWriter

Work for Week 3: class Histogram

The way we have made a histogram is not elegant and there is a much better way of doing it. We will take *a first look* at the central idea of Object Orientated (OO) programming and construct a separate class, **Histogram** which we will invoke from another class (our program) **MakeHistograms**. The new class stores our histogram. There are thus TWO classes, with an overall structure as follows:

```
class Histogram
{
      // constructor method
      public Histogram ( optional parameter list  )
      {....  }
      public String method1 ( optional paremeters)
      { ....  }
      public int method2 ( parameters )
      { ....  }
}
```

```
class MakeHistograms
{
     static methods ( )
     {  ...  }
     public static void main(      )
     {    .....
          Histogram myhisto1  = new Histogram ( optional parameter list);
          .....
     }
}
```

p.t.o.

Work for Week 3: class Histogram

1) This method of making a program is called *Encapsulation* as we are making an *abstract data type* called **Histogram**.

2) Notice only one of the classes (**MakeHistograms**) contains a main method. *This is the method which runs when the program starts.* In the main method, one creates instances (often called object) of a Class with the command:

   `Classname myinstancename = new Classname( optional parameters)`

   This line causes the *constructor instance method* to run and create a copy of the object in memory, with its initial values set up.

3) Once an instance is made its methods can be accessed using

   `myinstancename.methodname( )`

   e.g. we add data into the histogram with the command **`hist1.fillh(nextone);`**

   here

   - (a) **`hist1`** is the reference variable pointing to object Histogram
   - (b) **`fillh`** is the *instance method* in class Histogram.
   - (c) **`nextone`** is the next random number.

Work for Week 3: class Histogram

Type in and set up the classes **MakeHistograms** and **Histogram ,** adding **instance methods** to class **Histogram** that let the user get the **underflows** and the **overflows** of the histogram as well as the statistical error in a given bin.

Explain in your report what the constructor method does in general and in the case of the ***Histogram*** *class.*

[1.5]

Add the method **writeToDisk** from last weeks exercises as a further instance method to this class. Revise the parameter removing parameters that are no longer needed. Load the output from **writeToDisk** in Excel and plot the already defined histogram with random numbers between 0.4 and 0.9. Make sure the x-axis of your histograms correctly show the bin-centre in x and not the bin number.

[1.5]

Add the method **gauss** to class **MakeHistograms**, then create **a second instance** of **Histogram** in the **main** method of **MakeHistograms**, to make a histogram of 2000 numbers following a Gaussian distribution which has 0 as it central value and a width of 0.5. Make a Histogram in Excel. Use 20 bins over a range from -1.0 to 1.0 .

Explain in your report why the code in the method ***gauss*** *gives you a (nearly) Gaussian distribution (look up "central limit theorem") .*

[1.5]

Use your code make a third histogram of the numbers ***D*** produced when you take a random number in the range $0 < r < 1$ and define $D = -C*\ln(r)$ (note the –ve sign). Take $C = 15$ and work out an appropriate range and binning for this histogram. Make a Histogram in Excel.

[1.5]

Work for Week 3: class MakeHistograms

```
import java.io.*;
import java.util.Random; // notice this..needed to load the class Random.
class MakeHistograms
{
     static BufferedReader keyboard = new BufferedReader (new InputStreamReader(System.in))
     ;
     static PrintWriter screen = new PrintWriter( System.out, true);
     static Random  value = new Random(); //This line must only be used once in any program!
     public static void main (String [] args ) throws IOException
     {
          // It helps in debugging to force the same random numbers to be used each time.
          //long seed = 38945628; // choose some large integer to be the seed
          //value.setSeed(seed); // use the method "setSeed" in Class Random
          int trials;
          screen.print( "Input the number of random numbers to generate "); screen.flush();
          trials = new Integer(keyboard.readLine()).intValue();
          // create an instance of the Class Histogram
          Histogram hist1 = new Histogram("Random numbers",20, 0.4, 0.9);
```

p.t.o.

Work for Week 3: class MakeHistograms (cont.)

```
          for ( int goes=1; goes <= trials; goes++)
          {
               double nextone = value.nextDouble();
               hist1.fillh(nextone); /* put this into the histogram using the instance
                              method fillh in the Object with reference variable hist1 */
          }
          //histogram has been filled. Show the contents on the screen.
          //Also, add up the contents of the bins to see if the sum equals trials
          screen.println( "Title of histogram =  " + hist1.getTitle() );
          double sum = 0;
          // find how many bins using the instance method "getSize()"
          int numberbins= hist1.getSize();
          for (int bins =0; bins <= numberbins-1; bins++) //
          {
               screen.println(  hist1.getContent(bins) + "\t");
               sum = sum + hist1.getContent(bins);
          }
          //hist1.writeToDisk(c:\\mydata.csv"); // method doesn’t exist yet
          screen.println(" the number of trials = " + trials + " ,
                         the sum of the contents =" + sum );
     }
}
```

Work for Week 3: class Histogram

```
import java.io.*;
class Histogram
{
      // these variables have class scope. see Hubbardpage 197 for use of 'protected'
      protected double binsize, binlow, binhigh,
      protected String title;
      protected int SIZE, underflow, overflow;
      int[] hist;  // define an integer array to store the histogram

      // constructor method for the class Histogram
      public Histogram(String t, int S , double binlo, double binhi)
      {
            // store the parameters in local variables to be used later
            title = t;
            SIZE=S;
            binlow = binlo;
            binhigh = binhi;
            // calculate any variables that might be useful later.
            binsize = ( binhigh - binlow)/ (double) SIZE;
            hist =new int[SIZE];
            underflow=0;
            overflow=0;
      }
      //-------------------------------------
      // instance methods start here
      //-------------------------------------
      public int getSize() { return SIZE;}
      //----------------------------------------
```

p.t.o.

Work for Week 3:class Histogram (cont.)

```
    public void fillh( double x)
    {
        if( x > binlow && x < binhigh)
        {
            // update the correct bin
            int bin = (int) ( ( x - binlow)/binsize);
            hist[bin]++; // add 1 to the bin
        }
        else
        {
            if (x <= binlow ) underflow++;
            if (x >= binhigh) overflow++;
        }
    }
    //-------------------------------------
    public String getTitle()
    {
        // returns the title of the histogram to the user
        return  title;
    }
    //-------------------------------------
    public int getContent(int nbin)
    {
        // returns the contents on bin 'nbin' to the user
        return hist[nbin];
    }
}
```

Work for Week 3: gauss method

```
private static double gauss( double xmean, double sigma )
{
    double newGauss, sum;
    sum=0;
    for (int n=0 ; n<=11; n++)
    {
        sum=sum + value.nextDouble( );
    } // add up 12 random numbers
    newGauss = xmean + sigma*(sum -6);
    return newGauss;
}
```