PHYS 488

Modelling Physical Phenomena

Lecture 3

Jan Kretzschmar

Recap from week 2: *if* statement

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

```
if (boolean test){
   statements executed if the test is TRUE
} else {
   statements executed if the test is FALSE
}
```

else – blocks are always optional!
Can also chain or nest conditions, use with care and watch your curly brackets:

```
if (n > 0){
   screen.println(“n is greater than zero”);
   if (n == 42){
      screen.println(“n is 42”);
   }
} else if (n < 0){
   screen.println(“n is negative”);
} else {
   screen.println(“n is 0”);
}
```

Recap from week 2: loops

for-loop: compact way to execute instructions a fixed number of times, e.g. iterating over all values of an array

Initialize; Test condition; Increment

```
    for (int n = 0; n < hist.length; n++) {
        screen.println(" hist[" + n + "] = " + hist[n]);
    }
```

Loop Body

while-loop: more suitable, when the number of iterations is not know in advance

```
    int number = 0;
    while (number != 42) {
        screen.println("Say 42");
        number = new Integer(keyboard.readLine() ).intValue();
    }
```

Recap from week 2: *arrays*

Arrays useful for dealing with series of variables of the same type, for example the bins of a histogram, elements of a vector etc.

```
final int SIZE = 20;
int [] hist = new int[SIZE];
Screen.println(“Array has “ + hist.length + “ elements.”);
```

Very often we iterate over arrays with a for-loop. There is a (rather new) syntax to make this easier, although at the expense of some flexibility.

```
   for (int n = 0; n < hist.length; n++) {
        screen.println(hist[n]);
   }
   for (int element : hist) {
        screen.println(element);
   }
```

Note: the use of the second form is completely optional, if unsure, stick to the more general form.

Recap from week 2: *cast* statements

A ***cast statement*** is sometimes needed to convert between different data-types.

Obviously, the computer will not perform any magic here and only make conversions that may seem rather trivial.

E.g. often we wish to convert floating point to integer values or vice-versa.

```
int bin = (int)((randomNumber-binlow)/binsize));
int a = 3;
int b = 4;
double x = a/b;
double y = (double)a/(double)b;
screen.println(x);
screen.println(y);
```

What is the output of this fragment?

Some hints on good programming practice

Indent each new block in a Java class and pair the **{ }** in a systematic way. Helps to find misplaced, missing or extra brackets.

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

Avoid "**hard-coding**" numbers in your code, instead use variables ("final" if appropriate)

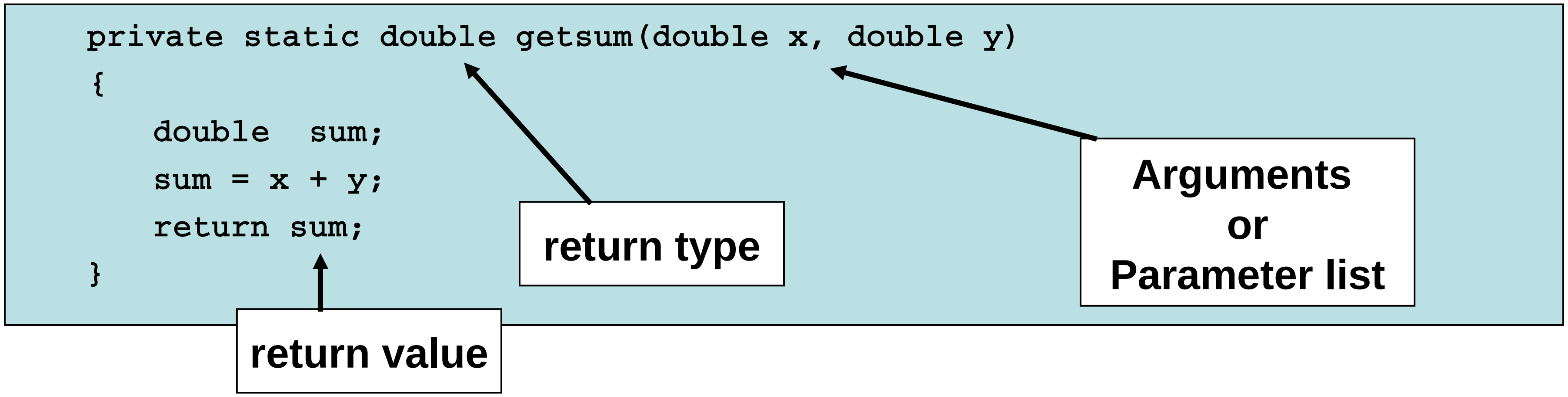
Avoid:

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

Better:

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

Recap week 2: Methods



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

Critical to structure the program into a number of **independent *methods*** that perform specific sub-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***.

The *scope* of variables

```
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**. The names of variables have **NO** influence that – two variables with the same name in different methods are independent. In general, variables “live” only inside their set of curly brackets.

In the parameter list of a method **only the values are passed** (copied), hence the variable of the calling method is always unchanged. E.g. here the variables “*first*” and “*second*” cannot be modified either by accident or by design from inside the method “*getsum*”. This is an important feature that helps to prevent mistakes.

The *scope* of variables

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);
      .....
```

Object Oriented Programming and Classes

Using **methods** to make the program code **modular** is an important first step.

In Object Oriented Programming, we go one step further to bundle **methods** with the **data** to **classes.**

The idea is, that methods and data often belong together:

- A Lorentz **four-vector** can be represented by **four floating point numbers** and we can **calculate its mass**, **add it to other four-vectors** etc.
- We can think of other examples, where the data is represented by (four) **floating point numbers,** e.g. the balance of four accounts. We could **calculate the total** simply by adding up all numbers.

Even if the data structures are similar or the same, clearly we would not want to use the same methods on this very different data!

Example: Four-vector calculations (from week 2)

```
private static double momentum(double [] vec)
{
    double p = 0.;
    for (int i = 1; i < 4; i++) {
        p += vec[i]*vec[i];
    }
    p = Math.sqrt(p);
    return p;
}

private static double mass(double [] vec)
{
    double p = momentum(vec); // reuse momentum method!
    return Math.sqrt(vec[0]*vec[0] - p*p);
}
….
public static void main (String [] args ) throws IOException
{
    double [] vec1 = new double [4];
    ...
    screen.println("Particle 1");
    screen.println("Enter E, px, py, pz in GeV");
    for (int i = 0; i < 4; i++) {
        vec1[i] = new Double(keyboard.readLine()).doubleValue();
    }
    ...
    screen.println("Particle 1: p = " + momentum(vec1) + " GeV, m = " + mass(vec1) + " GeV");
    …
}
```

Data

Methods

Example: a Four-vector class

```
class FourVector
{
    private double [] vec;

    public FourVector()
    {
        vec = new double[4];
    }
    ...
    public double momentum()
    {
        double p = 0.;
        for (int i = 1; i < 4; i++) {
            p += vec[i]*vec[i];
        }
        p = Math.sqrt(p);
        return p;
    }

    public double mass()
    {
        double p = momentum(); // reuse momentum method!
        return Math.sqrt(vec[0]*vec[0] - p*p);
    }
}
```

Class data ("field")

Constructor (default initialisation)

Class methods

- Data is **private** – can only be manipulated (correctly) through class methods
- Methods work on the data of a specific object – empty/short argument lists
- No **main**-method – a pure "**tool collection**" for reuse!

Example: Use the Four-vector class

```
import java.io.*;
class UseFourVector
{
    static BufferedReader keyboard = new BufferedReader (new InputStreamReader(System.in)) ;
    static PrintWriter screen = new PrintWriter( System.out, true);

    public static void main (String [] args ) throws IOException
    {
        FourVector vec1 = new FourVector();
        FourVector vec2 = new FourVector();
        ...
        screen.println("Particle 1");
        vec1.Input(screen, keyboard);
        screen.println("Particle 2");
        vec2.Input(screen, keyboard);
        ...
        screen.println("Particle 1: p = " + vec1.momentum() + " GeV, m = " + vec1.mass() + " GeV");
        screen.println("Particle 1: p = " + vec1.momentum() + " GeV, m = " + vec1.mass() + " GeV");

        FourVector vec3 = vec1.add(vec2);
    }
}
```

- Create (*instantiate*) objects of class FourVector with name vec1, vec2; for each an (internal) array is created and initialised (constructor)
- Call class methods: `vec1.momentum()` – note difference to `momentum(vec1)`
- Data is private – “hidden” (encapsulated), no direct access like `vec1[0]` = … – need to write additional methods to read/write data `vec1.Input(...)`

Example: more Four-vector class methods

```
    public double GetElement(int i)
    {
        return vec[i];
    }
    public void SetElement(int i, double value)
    {
        vec[i] = value;
    }
    public void Print(PrintWriter screen)
    {
        screen.println("E = " + GetElement(0) + " GeV, px = " + GetElement(1)
                       + " GeV, py = " + GetElement(2) + " GeV, pz = " + GetElement(3));
    }
    public void Input(PrintWriter screen, BufferedReader keyboard) throws IOException
    {
        screen.println("Enter E, px, py, pz in GeV");
        for (int i = 0; i < 4; i++) {
            SetElement(i, new Double(keyboard.readLine()).doubleValue());
        }
    }
    public FourVector add(FourVector vec2)
    {
        FourVector sum = new FourVector();
        for (int i = 0; i < 4; i++) {
            sum.SetElement(i, GetElement(i) + vec2.GetElement(i));
        }
        return sum;
    }
```

- Additional methods to read/write data creates some overhead, but helps to "abstract" from details of how data is stored
- Can formulate some operations in short and easily recognisable way, e.g. addition: `FourVector vec3 = vec1.add(vec2)`

Example: Output devices as a class

Object Oriented Programming supports “abstraction” from concrete implementation details in an intuitive way.

Example: we often send some text (string) to an output device, without worrying about details – text may end up on screen or text file.

The code looks very suggestive: `outputDevice.println(string);`

Writes to screen

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

Create instance
Use class method

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();
     }
}
```

Create instance
Use class methods

Histogram program (from week 2)

```
class GenerateHistogram
{
    private static void writeToDisk(int [] hist, double [] hist_errors,
                                    double low, double dx, int underflow, int overflow, String filename) ...
    {
       ...
    }

    public static void main (String [] args ) throws IOException
    {
        final int SIZE = 20;
        final double binlow  = 0.4;
        final double binhigh = 0.9;

        int [] hist = new int[SIZE];
        int underflow = 0;
        int overflow = 0;
        double [] hist_errors = new double[SIZE];
        ...
        for (int i = 0; i < trials; i++) {
            double randNumber = value.nextDouble();

            if (randNumber < binlow) {
                underflow++;
            } else if (randNumber >= binhigh) {
                overflow++;
            } else {
                // calculate which bin of the array should be increased by 1
                int bin = (int) ( (randNumber - binlow)/binsize );
                hist[bin]++; // add 1 to the location in the array hist
            }
        }
        ...
        writeToDisk(hist, hist_errors, binlow, binsize,underflow, overflow, "test.csv");
```

Class Histogram

The way we created and used a histogram last week is not elegant, although the "writeToDisk" method was a good start. The data structures that define the histogram properties and content and the code that manipulates the data correctly are scattered in the main program. Just imagine how difficult it would be to create and manipulate several additional histograms!

In the exercises you will construct a separate **Histogram** class, which can be used from the main program **MakeHistograms**. The overall structure will look like:

```
class Histogram
{
      private int [] data;
      ...
      public Histogram (optional parameter list) // constructor
      { ....  }
      public void fill (double value)
      { ....  }
      public void writeToDisk (String filename)
      { ....  }
      public int anotherMethod (parameters)
      { ....  }
}
```

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