6. Functions and Libraries

Prof. Dr. Charles Wüthrich
B.Sc. Francesco Andreussi,
CoGVis/MMC, Faculty of Media
Bauhaus-University Weimar

Introduction

- When writing programs, until now we have written our code as a part of the main program.
- To organize the programs better, it makes sense to subdivide the tasks into smaller self contained chunks, i.e. functions or, in Java, *static methods*
- For example, you might want
  - your main program to deal with user input,
  - While the actual drawing is done in a separate part

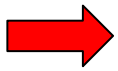
Functions (distance)

```java
// ClicksDistance: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class Distance {
    public static void main(String[] args) {
        double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
        double distan;

        x1=Double.parseDouble(args[0]);
        y1=Double.parseDouble(args[1]);
        x2=Double.parseDouble(args[2]);
        y2=Double.parseDouble(args[3]);

        distan=Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
        System.out.println(distan);
    } // main
}  // Distance
```

Functions (distance)

- To make it a bit simpler, I can separate the calculation from the input and output processing

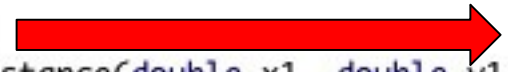
```
// ClicksDistance: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class Distance {
    public static void main(String[] args) {
        double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
        double distan;

        x1=Double.parseDouble(args[0]);
        y1=Double.parseDouble(args[1]);
        x2=Double.parseDouble(args[2]);
        y2=Double.parseDouble(args[3]);

        distan=Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
        System.out.println(distan);
    } // main
}  // Distance
```



Make this a method

Functions (distance)

```
// ClicksDistanceFun: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class DistanceFun {
    public static double distance(double x1, double y1, double x2, double y2){
        double distance,xdiff,ydiff;

        xdiff=x2-x1;
        ydiff=y2-y1;
        distance=xdiff*xdiff+ydiff*ydiff;
        distance=Math.sqrt(distance);
        return distance;
    }

  public static void main(String[] args) {
      double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
      double distan;

      x1=Double.parseDouble(args[0]);
      y1=Double.parseDouble(args[1]);
      x2=Double.parseDouble(args[2]);
      y2=Double.parseDouble(args[3]);

      distan=distance(x1,y1,x2,y2);
      System.out.println(distan);
  } // main
} // DistanceFun
```



Functions (distance)

```
// ClicksDistanceFun: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class DistanceFun {
    public static double distance(double x1, double y1, double x2, double y2){
        double distance,xdiff,ydiff;

        xdiff=x2-x1;
        ydiff=y2-y1;
        distance=xdiff*xdiff+ydiff*ydiff;
        distance=Math.sqrt(distance);
        return distance;
    }

  public static void main(String[] args) {
      double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
      double distan;

      x1=Double.parseDouble(args[0]);
      y1=Double.parseDouble(args[1]);
      x2=Double.parseDouble(args[2]);
      y2=Double.parseDouble(args[3]);

      distan=distance(x1,y1,x2,y2);
      System.out.println(distan);
  } // main
} // DistanceFun
```

Definition

Arguments

Return value

Method call

Functions (distance)

- When the `main` is executed, when the program encounters the function call it executes the code in the function.
- It passes to the function the values of the parameters
- The static function receives these values, processes its instructions, and when it ends, it passes back to the calling program the variable following the keyword `return`
- So

```
return distance;
```

  means:
  - When you go back to the calling program, in this case main, give back the content of the variable distance to the calling program

Functions (distance)

```
// ClicksDistanceFun: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class DistanceFun {
    public static double distance(double x1, double y1, double x2, double y2){
        double distance,xdiff,ydiff;

        xdiff=x2-x1;
        ydiff=y2-y1;
        distance=xdiff*xdiff+ydiff*ydiff;
        distance=Math.sqrt(distance);
        return distance;
    }
```

Signature

Return value, must be same type as in signature

```
  public static void main(String[] args) {
      double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
      double distan;

      x1=Double.parseDouble(args[0]);
      y1=Double.parseDouble(args[1]);
      x2=Double.parseDouble(args[2]);
      y2=Double.parseDouble(args[3]);

      distan=distance(x1,y1,x2,y2);
      System.out.println(distan);
    } // main
}  // DistanceFun
```

Method call, same variable types as in signature

Functions (distance)

- Properties of static methods:
  - Multiple arguments (you can pass as many as you want)
  - Multiple methods (a Java program can have as many methods as necessary)
  - Overloading: methods with different signatures are considered different methods
  - Multiple returns: you can put as many return statements as you like: the method will return to the calling method when it reaches the first one it encounters
  - Single return values: a method gives to the calling program a single value (but also `void` is allowed, i.e. no return values)

Functions (distance)

- Scope of a variable
  - Is the part of a program that can refer to a certain variable by name.
  - When a static method declares a variable, it can be only referred to in that block, and not elsewhere in the program.
  - So new variables in a method live only inside that method.

```
// ClicksDistanceFun: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class DistanceFun {
    public static double distance(double x1, double y1,
                                  double x2, double y2){
        double distance,xdiff,ydiff;

        xdiff=x2-x1;
        ydiff=y2-y1;
        distance=xdiff*xdiff+ydiff*ydiff;
        distance=Math.sqrt(distance);
        return distance;
    }
```

Cannot be reached from main

```
    public static void main(String[] args) {
        double x1=0.0,y1=0.0,x2=0.0,y2=0.0;
        double distan;

        x1=Double.parseDouble(args[0]);
        y1=Double.parseDouble(args[1]);
        x2=Double.parseDouble(args[2]);
        y2=Double.parseDouble(args[3]);

        distan=distance(x1,y1,x2,y2);
        System.out.println(distan);
    } // main
} // DistanceFun
```

Functions (distance)

- Notice! When passing the arguments the function makes for its scope COPIES of the arguments: modifications will not be seen by the calling function
- This is called *call by value*
- However, if you pass an array as an argument then the modifications to it will be seen by the caller
  - Reason for this: when an array is passed, only the reference to where it is in memory is passed, but no elements are copied.

```
// swaps array elements i and j
public static void exchange(String[] a, int i, int j) {
    String temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Functions (distance)

- Now that we know functions, we can as well use it for our two points program
- We capture the mouse position for 2 clicks, and compute the distance of the points we clicked

- We can do this by using the same DistanceFun we used in the simple program, and copying it at top of the two points program

Functions (distance)

- Now that we know functions, we can as well use it for our two points program
- We capture the mouse position for 2 clicks, and compute the distance of the points we clicked

- We can do this by using the same DistanceFun we used in the simple program, and copying it at top of the two points program

Function example

- Now that we know a distance functions, we can use it for example, towrite a program that
  - opens a window
  - waits for two mouse clicks
  - computes the distance between the clicked points
- We will base the example on the Mouseclick function seen in the last lecture
- However, this time we need two points for computing the distance, and to wait for the user to give 2 points
- Waiting for two points is quite tricky:
  - Remember if you clicked a first one
  - Draw line when you have two (so remember if you clicked a second)

Function example

- We can start here:

```
// MouseClick: Draws a circle at mouse position
// usage: java MouseClick

public class MouseClick {
   public static void main(String[] args) {
      double x1,y1;
      boolean ispressed=false;
      double radius; // radius of circle

      StdDraw.setPenColor(255,0,0);
      radius=0.01;

      while (true) {
        ispressed=StdDraw.mousePressed();
        if(ispressed){
          x1=StdDraw.mouseX();
          y1=StdDraw.mouseY();
          StdDraw.filledCircle(x, y, radius);
          ispressed=false;
        }
      }
   } // main
} // MouseClick
```

Click two points

```
// ClicksLine: Draws line between two points
// usage: java ClicksLine
public class ClicksLine {
    public static void main(String[] args) {
        double x1=0.0,y1=0.0,x2=-1.0,y2=-1.0;
        double mousex=-2.0, mousey=-2.0;
        boolean ispressed=false;
        boolean p1filled=false;
        double radius;     // radius of circle

        StdDraw.setPenColor(255,0,0);
        radius=0.01;

        while (true) {
           ispressed=StdDraw.mousePressed();
           if(ispressed){
               while(true){
                   // check when the user releases the mouse
                   ispressed=StdDraw.mousePressed();
                   if(!ispressed) break;
               }
               // at this point the user released the mouse:
               // now fetch its coordinates
               mousex=StdDraw.mouseX();
               mousey=StdDraw.mouseY();
```

- Notice the code at the bottom:
  - `While(true)` means FOREVER
  - There are two of them
  - First one is the main loop for the window drawn
  - Second one does nothing until the mouse button is released
  - When it is released, it reads the coordinates of the mouse
  - So when I finish this part I have the x and y of the mouse

Click two points

```
// have we already filled the first point?
if(p1filled == false){
      // then copy the mouse coordinates to it
      x1=mousex;
      y1=mousey;
      // draw a circle at this point
      StdDraw.filledCircle(x1, y1, radius);
      System.out.println("X1: "+ x1 + "," + y1);
      // now remember that the first point was filled
      p1filled=true;
}
```

- Now I have my mouse:
  to draw a line I need 2 points

  Notice the IF at the start?
  - `P1filled` is a boolean control variable:
    `true` if I already have 1pt
    `false` if not
  - If I don't have 1st pt
    we copy the captured mouse position to x1 and y1
  - AND we set `P1filled` to true, so next time we don't fill again 1st point

Click two points

```
            else{
                // if we are here, the first point was filled:
                // so mousex and mouse y have the coordinates
                // of the secod point
                x2=mousex;
                y2=mousey;
                // Draw second point
                StdDraw.filledCircle(x2, y2, radius);
                System.out.println("X2:"+ x2 + "," + y2);
                // Draw Line joining the points
                StdDraw.setPenColor(0,0,255);
                StdDraw.line(x1,y1,x2,y2);
                StdDraw.setPenColor(255,0,0);

                // now we have drawn, so reset control variable
                p1filled=false;
                // reset also mouse press control ispressed
                ispressed=false;
            } // else

        }  // if ispressed
        // StdDraw.pause(10);
    } // while
  } // main
} // ClicksLine
```

- The else is for filling the 2nd point and draw
- Now we know 1st point is fillled
- If we are here,we have the mouse coords. For the 2nd point
- Draw the circle at the point

- We have both points, so we draw a line between them
- Finally, we clear two control variables:
  - `p1filled` for `p1`
  - `ispressed` for mouse press

Click two points

- Why did we use the mouse position when the mouse is released and not when the mouse is pressed?
- Because a user presses the mouse slowly compared to the speed of your computer
- So you get too many pressed mouse events which makes it difficult to control what is being filled where
- The user releases the mouse only once!

Click two points (distance)

- And the distance of the two points?
- Easy! Remember the distance function we did b4?

```
// ClicksDistanceFun: Computes distance between two points
// usage: java Distance x1 y1 x2 y2

public class DistanceFun {
    public static double distance(double x1, double y1,
                                  double x2, double y2){
        double distance,xdiff,ydiff;

        xdiff=x2-x1;
        ydiff=y2-y1;
        distance=xdiff*xdiff+ydiff*ydiff;
        distance=Math.sqrt(distance);
        return distance;
    }
```

- We simply add (copy) it after the
  `public class clickline{`
  line at the top

```
// compute and print the length
length=distance(x1,y1,x2,y2);
System.out.println("length: "+ length);
```

- And use it for computing (and then printing) the distance after having drawn the line

Reusing code

- For the distance calculations, we did not have to write ANY new code!
- We can REUSE what we did before!
- So, if when you develop your code, try to make it *modular*:
  - Reusable chunks
  - Encapsulated functions you can reuse when you write other code

Using static methods elsewhere

- For referring to a static method of one class in another class
- You must make both classes accessible to Java
  - Simplest way: putting them in the same directory
- Then you can refer the function as

  ```
  classname.functionname
  ```

  the dot says the function is a method of classname

- For example, in our proram we can avoid to paste the distance function in our code and write:

  ```
  length=DistanceFun.distance(x1,y1,x2,y2);
  ```

- …. Which is EVEN BETTER!

Libraries

- Code designed to reuse in other programs is called a *library*
- Some terminology must be introduced:
  - Clients: a client is a program using a library
  - API (Application Programming Interface): a description of the methods available in a library with their usage detail
    for example, the signatures of the functions
  - Implementation: the java code implementing the methods of a specific API
  - Example: implement the following API

public class Matrix

double dot(double[] a, double[] b)	*vector dot product*
double[][] multiply(double[][] a, double[][] b)	*matrix–matrix product*
double[][] transpose(double[][] a)	*transpose*
double[] multiply(double[][] a, double[] x)	*matrix–vector product*
double[] multiply(double[] x, double[][] a)	*vector–matrix product*

Recursion

- Now that we know more about functions, we can introduce an elegant concept called *recursion* that is often used in programming.
- In recursion, a method calls itself.
- This in Java is perfectly legitimate, and can be used to perform special computations, for example for simplifying computations until the computation is simple.
- When using recursion, one must make sure that the recursive call of the function ends at some point
  - otherwise the program makes an infinite loop, calling itself forever and ever.

Recursion: Example

- Perhaps the easiest example of recursion is given by computing the factorial of a number
  - The factorial of a number N, indicated as N!, is defined as the multiplication of all numbers smaller or equal to N

    N!=N * (N-1) * (N-2) * … * 2 * 1
- There are two ways of computing the factorial of a number:
  - Through a loop
  - Through recursion, which is particularly elegant
- We can observe that
  - N!= N * (N-1)!
  - (N-1)! = (N-1) * (N-2)!

    ...
  - 1! =1

Recursion: Example

- So… to compute the factorial of a number N one must
  - Multiply N with the factorial of N-1 until we reach 1

```
public static long factorial(long n) {
   if (n == 1) return 1;
   return n * factorial(n-1);
}
```

  - Simple and elegant!
- Nota Bene: the first line absolutely important!
  - If not there, the recursion continues FOREVER!

Recursion: Example

- Now the whole class computing our factorial allowing us to test:

```
public class Factorial{
   public static long factorial(long n) {
     if (n == 1) return 1;
     return n * factorial(n-1);
   } // factorial

   public static void main(String[] args) {
      long f;
      f=Long.parseLong(args[0]);
      f=factorial(f);
      StdOut.println("factorial="+f);
   } // main
} // Factorial
```

  - Executing:

```
> java Factorial 6
720
```

Factorial: Overflow

- Factorials become very quickly huge numbers!
  - Executing:
    ```
    > java Factorial 20
    factorial=2432902008176640000
    ```
  - But:
    ```
    > java Factorial 21
    factorial=-4249290049419214848
    ```
  - This is impossible!
    Cannot be a negative number!

  - What happened is called *variable overflow*:
    multiplying `21 * 2432902008176640000` is a number which is too big for a double variable!
  - The result of the multiplication is garbage!

Recursion: a 2nd example

- Let’s get away from numbers, and do recursive visuals

Recursion: a 2nd example

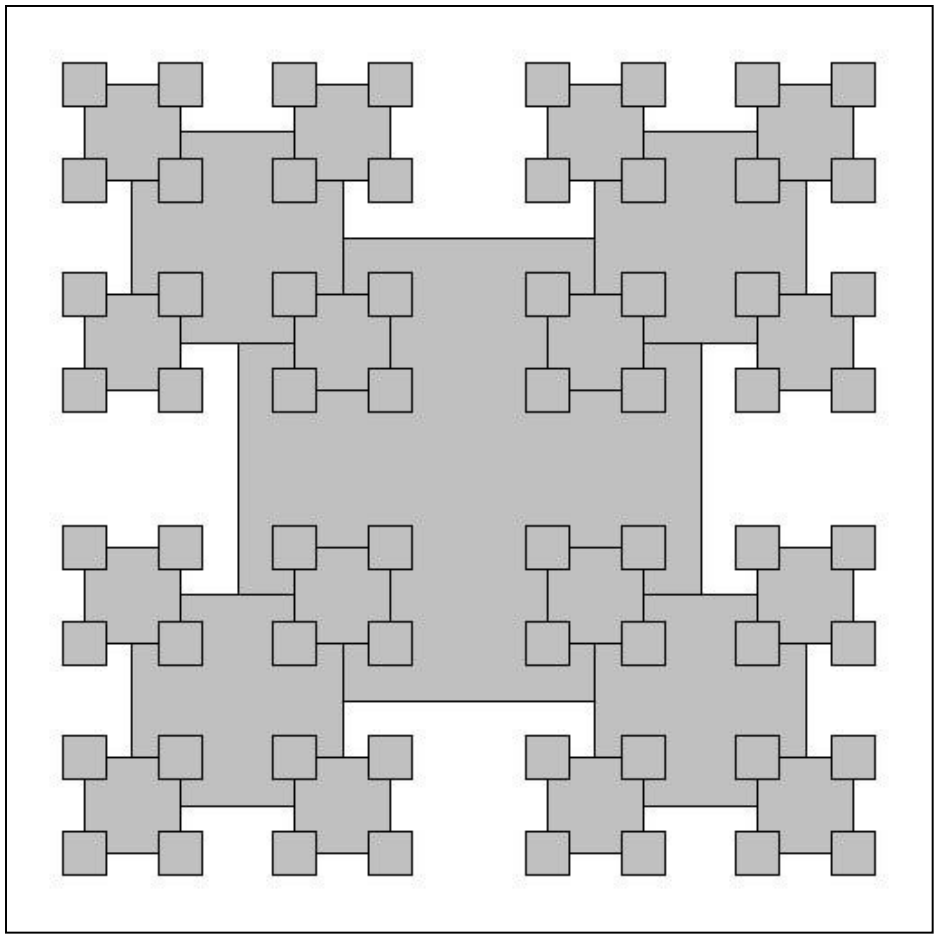
```
public class RecursiveSquares {

   // plot a square, centered on (x, y) of the given side length
   // with a light gray background and black border
   public static void drawSquare(double x, double y, double size) {
      StdDraw.setPenColor(StdDraw.LIGHT_GRAY);
      StdDraw.filledSquare(x, y, size/2);
      StdDraw.setPenColor(StdDraw.BLACK);
      StdDraw.square(x, y, size/2);
   }
   // plot an order n recursive squares pattern
   // centered on (x, y) of the given side length
   public static void draw(int n, double x, double y, double size)
        {
      if (n == 0) return;

      drawSquare(x, y, size);

      // 2.2 ratio looks good
      double ratio = 2.2;

      // recursively draw 4 smaller trees of order n-1
      draw(n-1, x - size/2, y - size/2, size/ratio); // lower left
      draw(n-1, x - size/2, y + size/2, size/ratio); // upper left
      draw(n-1, x + size/2, y - size/2, size/ratio); // lower right
      draw(n-1, x + size/2, y + size/2, size/ratio); // upper right
   }
```

Recursion: a 2nd example

```
   // read in an integer command-line argument n and plot
   // an order n recursive squares pattern
   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]);

      double x = 0.5, y = 0.5; // center of square
      double size = 0.5;  // side length of square
      draw(n, x, y, size);
   }

}
```

Recursion: a 2nd example

- Resulting execution (4 recursions)



End

+++ Ende - The end - Finis - Fin - Fine +++ Ende - The end - Finis - Fin - Fine +++