Functions / Static Methods

Why are functions so helpful?

Find Nearest Neighbour to a Central Point

Sequence of x-y point coordinates as arguments to program
class NearestNeighbourBad {
    public static void main(String[] args) {
        int N = args.length;
        if (N % 2 != 0) N--;
            // ignore final arg if odd number
        double[] points = new double[N];
        for (int i = 0; i < N; i++)
            points[i] = Double.parseDouble(args[i]);
        double[] centre = { points[0], points[1] };
            // first point is our centre
        System.out.printf("Centre lies at (%5.2f, %5.2f)\n", centre[0], centre[1]);
        double[] neighbours = new double[points.length - 2];
        for (int i = 2; i < points.length; i++)
            neighbours[i - 2] = points[i];
        double[] dists = new double[neighbours.length / 2];
        for (int i = 0; i < neighbours.length; i += 2) {
            // step over two at a time to get x and y
            double d1 = centre[0] - neighbours[i];
            double d2 = centre[1] - neighbours[i + 1];
            dists[i / 2] = Math.sqrt(d1*d1 + d2*d2);
        }
        for (int i = 0; i < dists.length; i++)
            System.out.printf("Distance to (%5.2f, %5.2f) is %5.2f\n",
                neighbours[(i*2)], neighbours[(i*2) + 1], dists[i]);
        double min = dists[0];
        for (int i = 1; i < dists.length; i++)
            if (dists[i] < min) min = dists[i];
        System.out.printf("Minimum distance to centre is %5.2f\n", min);
    }
}
Main Function for Nearest Neighbour

```java
public static void main(String[] args) {
    double[] points = parseArguments(args);
    double[] centre = getCentre(points);
    printCentre(centre);
    double[] neighbours = getNeighbours(points);
    double[] distances = calcDistances(centre, neighbours);
    printDistances(distances, neighbours);
    double minimum = calcMinimum(distances);
    printMinimum(minimum);
}
```

This is simply what we just developed plus some types and brackets.

Euclidian Distance between two Points

- Given some ‘special’ point \( p \), how close are various other points to \( p \)?
- Useful, for example, if trying to find the closest point to \( p \).
- Use Euclidean distance — restricted to 2D case, where \( p = (p_0, p_1) \) etc.:

\[
\text{dist}(p, q) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2}
\]

```java
public static double distance(double x0, double y0, double x1, double y1) {
    double d1 = x0 - x1;
    double d2 = y0 - y1;
    return Math.sqrt(d1*d1 + d2*d2);
}
```
Anatomy of a Java Function

public static double distance (double x0, double y0, double x1, double y1) {
    double d1 = (x0 - x1);
    double d2 = (y0 - y1);
    return Math.sqrt(d1*d1 + d2*d2);
}

Calling a Function

Literal arguments

double d = distance(3.0, 5.0, 14.25, 2.70);

Variable arguments

double p0 = 3.0;
double p1 = 5.0;
double q0 = 14.25;
double q1 = 2.70;

double d = distance(p0, p1, q0, q1);

Flow of Control with Functions

Schematic Structure of Program

public class PointDistance {
	public static double distance(double x0, double y0, double x1, double y1) {
	    ...
	}
}
	public static void main(String[] args) {
	    ...
	double dist = distance(p0, p1, q0, q1);
	    ...
	}
}
Functions provide a new way to control the flow of execution.

What happens when a function is called:
- Control transfers to the code in body of the function.
- Parameter variables are assigned the values given in the call.
- Function code is executed.
- Return value is assigned in place of the function call in the calling code.
- Control transfers back to the calling code.

Pass by Value

public class AddOne {
   public static void addOne(int num) {
      num++;
   }
   public static void main(String[] args) {
      int x = 0;
      addOne(x);
      System.out.println(x);
   }
}

Output
% java AddOne
0

Pass by Value: Arrays

Array types are reference types, so things work a bit differently with arrays as arguments:
- the array itself (and its length) cannot be changed;
- but its elements can be changed.
- So changing the value of the element of an array is a side-effect of the function.
Pass by Value: Arrays

```java
class AddOne {
    public static void addOne(int[] anArray) {
        anArray[0]++;
    }
    public static void main(String[] args) {
        int[] a = {0, 1};
        addOne(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

Output

```
% java AddOne
1
1
```

Solution - Using Functions

```java
class NearestNeighbour {
    public static double[] parseArguments(String[] args) {...}
    public static double[] getCentre(double[] points) {...}
    public static void printCentre(double[] centre) {...}
    public static double[] getNeighbours(double[] points) {...}
    public static double distance(double x0, double y0, double x1, double y1) {...}
    public static double[] calcDistances(double[] centre, double[] neighbours) {...}
    public static void printDistances(double[] dists, double[] neighbours) {...}
    public static double calcMinimum(double[] dists) {...}
    public static void printMinimum(double min) {...}
    public static void main(String[] args) {
        double[] points = parseArguments(args);
        double[] centre = getCentre(points);
        printCentre(centre);
        double[] neighbours = getNeighbours(points);
        double[] distances = calcDistances(centre, neighbours);
        printDistances(distances, neighbours);
        double minimum = calcMinimum(distances);
        printMinimum(minimum);
    }
}
```

Signature

The signature of a Java function consists of its name and its parameter list (number and type of parameters, in order).

Example signature

```java
max(int x, int y)
```

However, it’s often convenient to use the term more loosely to refer to the head of the function definition:

Example head of definition

```java
public static int max(int x, int y)
```

Return

- Return type of a function is stated in the header of the function declaration.
- A function declared void doesn’t return a value.
- Any function with a non-void return type rtype must contain a statement of the form
  ```java
  return returnValue;
  ```
  where the data type of returnValue matches the type rtype.
Cubes, 1

Cubes, version 1

```java
public class Cubes1 {
    public static int cube(int i) {
        int j = i * i * i;
        return j;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i <= n; i++) {
            System.out.println(i + " " + cube(i));
        }
    }
}
```

Output

```none
% java Cubes1 6
0 0
1 1
2 8
3 27
4 64
5 125
6 216
```

Cubes, 2

Cubes, version 2

```java
public class Cubes2 {
    public static int cube(int i) {
        int i = i * i * i;
        return i;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i <= n; i++) {
            System.out.println(i + " " + cube(i));
        }
    }
}
```

Compile-time error

```none
Duplicate local variable i
```

Cubes, 3

Cubes, version 3

```java
public class Cubes3 {
    public static int cube(int i) {
        int j = i * i * i;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i <= n; i++) {
            System.out.println(i + " " + cube(i));
        }
    }
}
```

Compile-time error

```none
This method must return a result of type int
```

Cubes, 4

Cubes, version 4

```java
public class Cubes4 {
    public static int cube(int i) {
        i = i * i * i;
        return i;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i <= n; i++) {
            System.out.println(i + " " + cube(i));
        }
    }
}
```

Don't do that!
Cubes, version 5

```java
public class Cubes5 {
    public static int cube(int i) {
        return i * i * i;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i <= n; i++) {
            System.out.println(i + " " + cube(i));
        }
    }
}
```

What are the Function Bodies for the Distance Problem?

```java
public static void main(String[] args) {
    double[] points = parseArguments(args);
    double[] centre = getCentre(points);
    printCentre(centre);
    double[] neighbours = getNeighbours(points);
    double[] distances = calcDistances(centre, neighbours);
    printDistances(distances, neighbours);
    double minimum = calcMinimum(distances);
    printMinimum(minimum);
}
```

Arguments

```java
public static double[] parseArguments(String[] args) {
    int N = args.length;
    if (N % 2 != 0) N--; // ignore final arg if odd number
    double[] p = new double[N];
    for(int i = 0; i < N; i++)
        p[i] = Double.parseDouble(args[i]);
    return p;
}
```

Centre

```java
public static double[] getCentre(double[] points) {
    // first point is our centre
    double[] c = { points[0], points[1] };
    return c;
}
```

```java
public static void printCentre(double[] centre) {
    System.out.printf("Centre lies at (%5.2f, %5.2f)\n", centre[0], centre[1]);
}
```

```java
public static void main(String[] args) {
    double[] points = parseArguments(args);
    ...
}
```
Neighbours

global static double[] getNeighbours(double[] points) {
    double[] n = new double[points.length - 2];
    // all except the first are neighbours
    for(int i = 2; i < points.length; i++)
        n[i - 2] = points[i];
    return n;
}
global static void main(String[] args) {
    ...
    double[] neighbours = getNeighbours(points);
    ...
}

Distance Calculation

global static double distance(double x0, double y0,
    double x1, double y1) {
    double d1 = x0 - x1;
    double d2 = y0 - y1;
    return Math.sqrt(d1*d1 + d2*d2);
}
global static double[] calcDistances(double[] centre, double[] neighbours) {
    double[] dists = new double[neighbours.length / 2];
    // step over two at a time to get x and y
    for(int i = 0; i < neighbours.length; i += 2)
        dists[i / 2] = distance(centre[0], centre[1],
    neighbours[i], neighbours[i + 1]);
    return dists;
}
global static void main(String[] args) {
    ...
    double[] distances = calcDistances(centre, neighbours);
    ...
}

Distance Print

global static void printDistances(double[] dists, double[] neighbours) {
    for(int i = 0; i < dists.length; i++)
        System.out.printf("Distance to (%5.2f, %5.2f) is %5.2f\n",
    neighbours[(i*2)], neighbours[(i*2) + 1], dists[i]);
}
global static void main(String[] args) {
    ...
    printDistances(distances, neighbours);
    ...
}

Minimum

global static double calcMinimum(double[] dists) {
    double min = dists[0];
    for(int i = 1; i < dists.length; i++)
        if (dists[i] < min) min = dists[i];
    return min;
}
global static void printMinimum(double min) {
    System.out.printf("Minimum distance to " +
    "centre is %5.2f\n", min);
}
global static void main(String[] args) {
    ...
    double minimum = calcMinimum(distances);
    printMinimum(minimum);
    ...
Functions and Modularity

Advantages of breaking a program into functions:

▶ decomposition of a complex programming task into simpler steps
▶ reducing duplication of code within a program
▶ enabling reuse of code across multiple programs
▶ hiding implementation details from callers of the function, hence
▶ readability, via well-chosen names.

Whenever you can clearly separate tasks within programs, you should do so.
Aim for methods of no more than 10-15 lines. Shorter is often good.

Code Duplication and Refactoring

```
public class Duplication0 {
    public static void main(String[] args) {
        String boyFirstName = "Jock";
        String boySecondName = "McIness";
        String boyName = boyFirstName + " " + boySecondName;
        int boyWeeklyPocketMoney = 2;
        int boySavingsTarget = 10;
        int boyWeeksToTarget = boySavingsTarget / boyWeeklyPocketMoney;
        System.out.print(boyName + " needs to save for ");
        System.out.println(boyWeeksToTarget + " weeks");
        
        String girlFirstName = "Jane";
        String girlSecondName = "Andrews";
        String girlName = girlFirstName + " " + girlSecondName;
        int girlWeeklyPocketMoney = 3;
        int girlSavingsTarget = 9;
        int girlWeeksToTarget = girlSavingsTarget / girlWeeklyPocketMoney;
        System.out.print(girlName + " needs to save for ");
        System.out.println(girlWeeksToTarget + " weeks");
    }
}
```

Output

```
% java Duplication0
Jock McIness needs to save for 5 weeks
Jane Andrews needs to save for 3 weeks
```
Q: What is ‘refactoring’?
A: the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure (M. Fowler)

Extract Method is only one of > 70 refactoring techniques listed by Fowler at http://www.refactoring.com/.
Modularity via Methods

Functionality has changed, so not a case of refactoring:

Output

% java Duplication4

**********************************************
Master Jock McInnes needs to save for 6 weeks

**********************************************
Miss Jane Andrews needs to save for 4 weeks

But wrapping code up in functions makes it much easier to localize modifications.

Summary: Using Functions / Static Methods

- Encourages good coding practices by emphasizing discrete, reusable methods;
- Encourages self-documenting code through good organization;
- When descriptive names are used, high-level methods can read more like a narrative, reducing the need for comments;
- Reduces code duplication.

Java functions:
- Take zero or more input arguments.
- Return at most one output value.
- Can have side effects; e.g., send output to the terminal.

What about recursive functions?
- Basic concepts same as in Haskell.
- One exercise (factorial) in this week’s labsheets.
- Refactoring improves the structure of code without changing the functionality of the application.
Learning Outcomes for this week

- Define and call static functions with zero or more parameters.
- Pass the return value of one function as the argument of another one.
- Declare and use local variables in functions.
- Use functions to modify an array ‘in-place’.
- Refactor a short program so that functionality is modularized into functions.

Reading

The order of topics in the Java Tutorial is different from the order of these slides, so at this point there isn’t an ideal match: the following reading anticipates some things we’ll cover later.

**Java Tutorial**

(Re)read pp33-37; then read pp87-99.

i.e., read the first part of Chapter 2 *Object-Oriented Programming Concepts* carefully now, but stop at *Inheritance*; and read the first part of Chapter 4 *Classes and Objects*, stopping at *Objects*. 