# Inf1B

## Functions aka Static Methods

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics

# Functions / Static Methods

# Why are functions so helpful?

Let's consider a program that helps children save pocket money towards a target.

# Why Functions

```
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");
    }
}
```

# Why Functions

## Output

```
$ java Duplication0
Jock McIness needs to save for 5 weeks
Jane Andrews needs to save for 3 weeks
```

# Why Functions

Lots of duplicate code in this implementation.

```
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");
    }
}
```

# Why Functions

```java
public class Duplication1 {

    public static String joinNames(String n1, String n2){
        return n1 + " " + n2;
    }

    public static void main(String[] args) {
        String boyName = joinNames("Jock", "McInnes");
        int boyWeeklyPocketMoney = 2;
        int boySavingsTarget = 10;
        int boyWeeksToTarget =  boySavingsTarget / boyWeeklyPocketMoney;
        System.out.print(boyName + " needs to save for ");
        System.out.println(boyWeeksToTarget + " weeks");

        String girlName = joinNames("Jane", "Andrews");
        int girlWeeklyPocketMoney = 3;
        int girlSavingsTarget = 9;
        int girlWeeksToTarget = girlSavingsTarget / girlWeeklyPocketMoney;
        System.out.print(girlName + " needs to save for ");
        System.out.println(girlWeeksToTarget + " weeks");
    }
}
```

extract new function

call new functio

# Why Functions

```
public class Duplication2 {

    public static String joinNames(String n1, String n2){
        return n1 + " " + n2;
    }

    public static int weeksToSavePocketMoney(int pocketMoney,
                                             int savingsTarget){
        return savingsTarget / pocketMoney;
    }

    public static void main(String[] args) {
        String boyName = joinNames("Jock", "McInnes");
        int boyWeeksToTarget =  weeksToSavePocketMoney(2, 10);
        System.out.print(boyName + " needs to save for ");
        System.out.println(boyWeeksToTarget + " weeks");

        String girlName = joinNames("Jane", "Andrews");
        int girlWeeksToTarget =  weeksToSavePocketMoney(3, 9);
        System.out.print(girlName + " needs to save for ");
        System.out.println(girlWeeksToTarget + " weeks");
    }
}
```

extract new function

call new function

# Why Functions

```java
public class Duplication3 {

    public static String joinNames(String n1, String n2){
        return n1 + " " + n2;
    }

    public static int weeksToSavePocketMoney(int pocketMoney,
                                             int savingsTarget){
        return savingsTarget / pocketMoney;
    }

    public static void printWeeksToSave(String name, int target){
        System.out.print(name + " needs to save for ");
        System.out.println(target + " weeks");
    }

    public static void main(String[] args) {
        String boyName = joinNames("Jock", "McInnes");
        printWeeksToSave(boyName, weeksToSavePocketMoney(2, 10));
        String girlName = joinNames("Jane", "Andrews");
        printWeeksToSave(girlName, weeksToSavePocketMoney(3, 9));
    }
}
```

extract new function

call new function

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself
- enabling reuse of code across multiple programs.

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself
- enabling reuse of code across multiple programs.

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself
- enabling reuse of code across multiple programs.

High-level motivation: readability and maintainability.

Get it right, keep it right.

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself
- enabling reuse of code across multiple programs.

High-level motivation: readability and maintainability.

Get it right, keep it right.

Concrete advice:

Whenever you can clearly separate tasks within programs, you should do so.

Aim for methods of no more than 10-15 lines. A function should do only one thing and that thing well.

# Modularity via Functions

Easier to change code broken down into functions.

```
public class Duplication4 {

    public static String joinNames(String n1, String n2){
        String title;
        if (n1 == "Jock") title = "Master";
        else title = "Miss";
        return title + " " + n1 + " " + n2;
    }

    public static int weeksToSavePocketMoney(int pocketMoney, int savingsTarget){
        double sweeties = 0.25;
        double reducedPocketMoney = pocketMoney * (1 - sweeties);
        return (int) (savingsTarget / reducedPocketMoney);
    }

    public static void printWeeksToSave(String name, int target){
        System.out.println();
        System.out.println("**********************************************");
        System.out.println(name + " needs to save for " + target + " weeks");
    }

    public static void main(String[] args) {
        String boyName = joinNames("Jock", "McInnes");
        printWeeksToSave(boyName, weeksToSavePocketMoney(2, 10));

        String girlName = joinNames("Jane", "Andrews");
        printWeeksToSave(girlName, weeksToSavePocketMoney(3, 9));
    }
}
```

# Modularity via Functions

**Output**

```
$ java Duplication4

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

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

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

# Taking a Closer Look

Let's calculate the Euclidean distance between two points.

# Euclidean Distance between two Points

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

$$dist(\mathbf{p}, \mathbf{q}) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2}$$

# Euclidean Distance between two Points

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

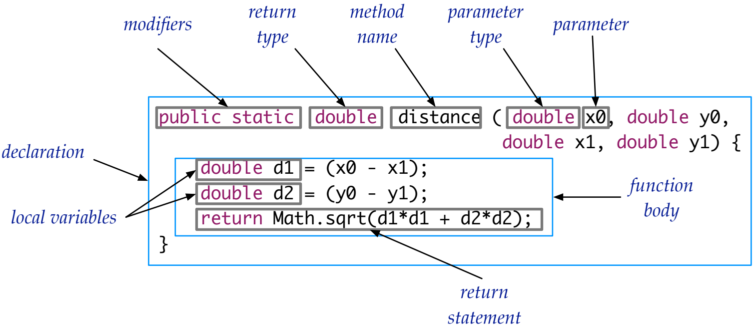
$$dist(\mathbf{p}, \mathbf{q}) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2}$$

```
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);
}
```

# Anatomy of a Java Function



# 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);

        ...

    }
}
```

# Flow of Control with Functions

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

- Pass by Value: parameter variables are assigned the values given by arguments to the call.
- The function only has access to the values of its arguments, not the arguments themselves.
- Consequently, changing the value of an argument in the body of the code has no effect on the calling code.

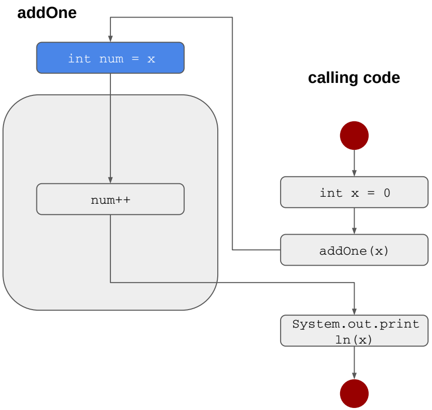
# Pass by Value

```java
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);
  }
}
```

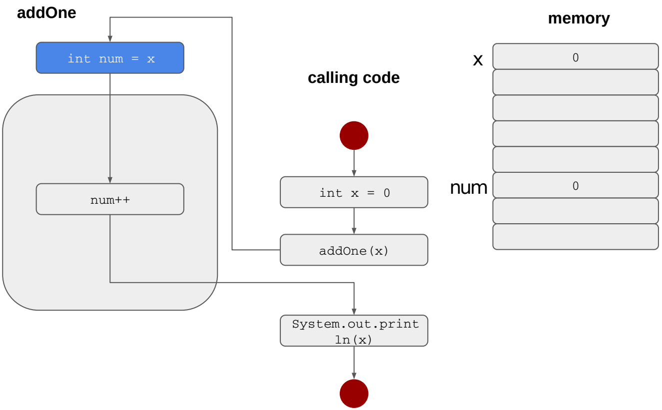
### Output

```
$ java AddOne
0
```

# Pass by Value



# Pass by Value



# Signature

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

## Example signature

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

# Signature

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

## Example signature

```
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

```
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

```
return returnValue;
```

where the data type of `returnValue` matches the type *rtype*.

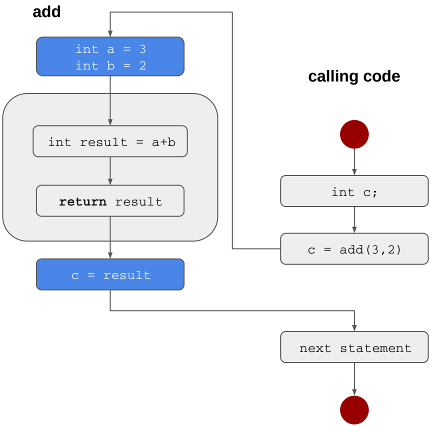
# Return

```
public class AddReturn {
  public static int add(int a, int b) {
    int result = a + b;
    return result;
}
  public static void main(String[] args) {
    int c = 0;
    c = add(3, 2);
    System.out.println(c);
  }
}
```

Output

```
$ java AddReturn
5
```

# Pass by Value



# 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.

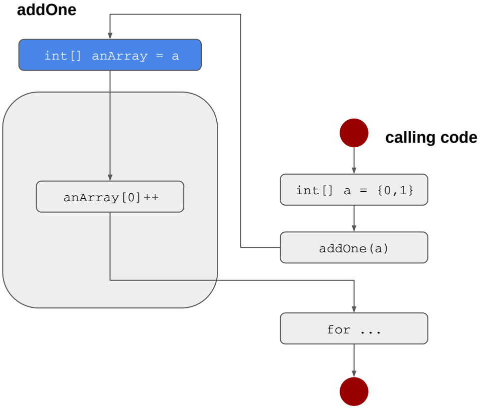
# Pass by Value: Arrays

```
public 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]);
    }
  }
}
```

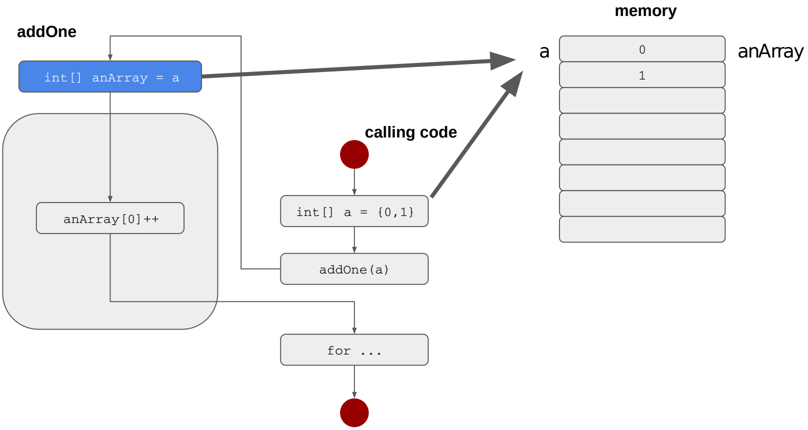
## Output

```
$ java AddOne
1
1
```

# Pass by Value: Arrays - Reference Types



# Pass by Value: Arrays - Reference Types



# Let's practise that

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it

# Cubes

Choose the correct instruction(s) to be placed in the body of the function such that the function will return the cube of a number.

# Cubes

Choose the correct instruction(s) to be placed in the body of the function such that the function will return the cube of a number.

```
public static int cube (int x) {
    // what goes in here?
}
```

1. `int i = x * x * x;`
2. `int x = x * x * x;`
   `return x;`
3. `return x * x * x;`
4. `x = x * x * x;`
   `return x;`

# Cubes

Choose the correct instruction(s) to be placed in the body of the function such that the function will return the cube of a number.

```
public static int cube (int x) {
    return x*x*x;
}
```

1. `int i = x * x * x;`
2. `int x = x * x * x;`
   `return x;`
3. `return x * x * x;`
4. `x = x * x * x;`
   `return x;`

# What does this print?

```
public static void swap(int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
  System.out.println(a + "␣" + b);
}

public static void main(String[] args) {
  int a = 2;
  int b = 5;
  System.out.println(a + "␣" + b);
  swap(a,b);
  System.out.println(a + "␣" + b);
}
```

# What does this print?

```
public static void swap(int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
  System.out.println(a + "␣" + b);
}

public static void main(String[] args) {
  int a = 2;
  int b = 5;
  System.out.println(a + "␣" + b);
  swap(a,b);
  System.out.println(a + "␣" + b);
}
```

**Output**

2 5
5 2
2 5

Only copies of the calling code's a and b are swapped in the function.

# What does this print?

```java
public static void swap(int[] a) {
    int tmp;
    tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
    System.out.println(a[0] + "␣" + a[1]);
}

public static void main(String[] args) {
    int[] a = {2, 5};
    System.out.println(a[0] + "␣" + a[1]);
    swap(a);
    System.out.println(a[0] + "␣" + a[1]);
}
```

# What does this print?

```java
public static void swap(int[] a) {
    int tmp;
    tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
    System.out.println(a[0] + " " + a[1]);
}

public static void main(String[] args) {
    int[] a = {2, 5};
    System.out.println(a[0] + " " + a[1]);
    swap(a);
    System.out.println(a[0] + " " + a[1]);
}
```

## Output

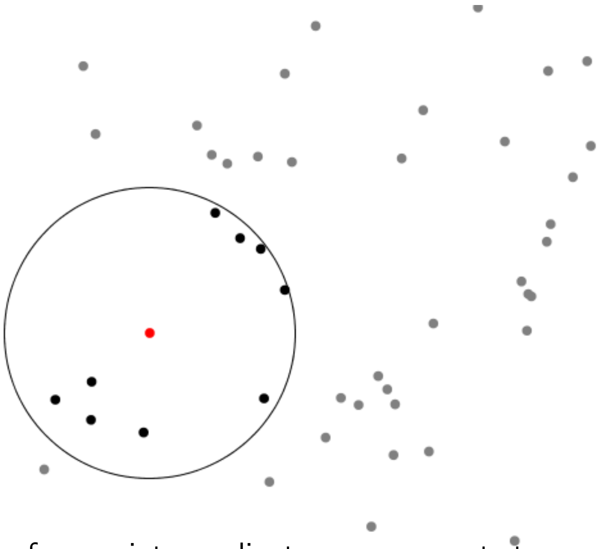
2 5
5 2
5 2

A reference to the calling codes array is copied and the original data is changed.

# Breaking Down Code as a Development Strategy

Let's find the nearest neighbour to a central point.

# Find Nearest Neighbour to a Central Point



Sequence of x-y point coordinates as arguments to program

# Solution

```
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++) // all except the first are
         neighbours
      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);
  }
}
```

# Easy, Right?

# Easy, Right?

Don't worry. Breaking this down into functions will make this much easier!

# What do we need to do?

# What do we need to do?

- parse arguments

# What do we need to do?

- parse arguments
- get centre

# What do we need to do?

- parse arguments
- get centre
- print centre

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours
- calculate distances

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

# What do we need to do?

- parse arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- get centre
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- get neighbours
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- calculate distances
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- distances $\leftarrow$ calculate distances $\leftarrow$ centre, neighbours
- print distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- distances $\leftarrow$ calculate distances $\leftarrow$ centre, neighbours
- print distances $\leftarrow$ distances
- calculate minimum
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- distances $\leftarrow$ calculate distances $\leftarrow$ centre, neighbours
- print distances $\leftarrow$ distances
- minimum $\leftarrow$ calculate minimum $\leftarrow$ distances
- print minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- distances $\leftarrow$ calculate distances $\leftarrow$ centre, neighbours
- print distances $\leftarrow$ distances
- minimum $\leftarrow$ calculate minimum $\leftarrow$ distances
- print minimum $\leftarrow$ minimum

Let's think about what we need for those steps.

The flow of the data

# What do we need to do?

- points $\leftarrow$ parse arguments $\leftarrow$ arguments
- centre $\leftarrow$ get centre $\leftarrow$ points
- print centre $\leftarrow$ centre
- neighbours $\leftarrow$ get neighbours $\leftarrow$ points
- distances $\leftarrow$ calculate distances $\leftarrow$ centre, neighbours
- print distances $\leftarrow$ distances
- minimum $\leftarrow$ calculate minimum $\leftarrow$ distances
- print minimum $\leftarrow$ minimum

Let's think about what we need for those steps.

The flow of the data

That is it!

# Main Function for Nearest Neighbour

```
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);
}
```

# Main Function for Nearest Neighbour

```
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.

All that is left to do is write some simple functions.

# Function Signatures / Headers

```
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);
  }
}
```

# Arguments

```
public static double[] parseArguments(String[] args) {
  int n = args.length;
  if (n % 2 != 0) n--; // ignore final arg if odd no
  double[] p = new double[n];

  for (int i = 0; i < n; i++) {
    p[i] = Double.parseDouble(args[i]);
  }
  return p;
}

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

# Centre

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

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

public static void main(String[] args) {
  ...
  double[] centre = getCentre(points);
  printCentre(centre);
  ...
}
```

# Neighbours

```
public 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;
}

public static void main(String[] args) {
  ...
  double[] neighbours = getNeighbours(points);
  ...
}
```

# Distance Calculation

```
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);
}


public 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;
}

public static void main(String[] args) {
    ...
    double[] distances = calcDistances(centre, neighbours);
    ...
}
```

# Distance Print

```
public 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]);
  }
}

public static void main(String[] args) {
  ...
  printDistances(distances, neighbours);
  ...
}
```

# Minimum

```
public 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;
}

public static void printMinimum(double min) {
    System.out.printf("Minimum␣distance␣to␣" +
            "centre␣is␣%5.2f\n", min);
}

public static void main(String[] args) {
    ...
    double minimum = calcMinimum(distances);
    printMinimum(minimum);
}
```

# Benefit of breaking a program into well-named functions

- decomposition of a complex task into simpler steps.
- hiding implementation details from the callers of a function
- reducing duplication of code within a program – Write Once, write DRY code where you **D**on't **R**epeat **Y**ourself
- enabling reuse of code across multiple programs.

High-level motivation: readability and maintainability.

Get it right, keep it right.

Concrete advice:

Whenever you can clearly separate tasks within programs, you should do so.

Aim for methods of no more than 10-15 lines. A function should do only one thing and that thing well.

# Summary: Using Functions / Static Methods

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.

# Summary: Using Functions / Static Methods

Structuring your code with methods has the following benefits:

- 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.

# Summary: Using Functions / Static Methods

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

# 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*.