Inf1-OP
Functions aka Static Methods
Timothy Hospedales, adapting earlier version by Perdita Stevens and Ewan Klein
School of Informatics
February 6, 2017

What's this “static method" business?

Behaviour in a Java system is defined by chunks of code called methods. Every method lives in a class. In Java, and related languages, you can think of a class as being both a module and a type. A class does several things:

- it's a way to split the code of the system up into cohesive chunks;
- it's a group of objects of that class;
- in its instance methods it defines what messages those objects understand and how they behave when they receive them;
- in its constructor(s) it defines how an object of the class is created;
- in its static methods it defines functions that implement behaviour related to the class, but not specific to any one object.

Static methods are rare compared to instance methods, but we start with them because they're more like Haskell functions.

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.

Examples we've already encountered:
- Java built-in functions: Math.random(), Math.abs(), Integer.parseInt().
- S&W I/O libraries: StdDraw.line().
- User-defined functions: main().

Haskell function: type declaration

```
max :: Int -> Int -> Int
```

Java function: type declaration

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

Java function: example with no return value

```
public static void printName(String fName, String lName)
```
Java functions: easy to write your own.

Distance between Points on Plane

- Given some ‘special’ point \( p \), how close are various other points to \( p \)?
- Useful, for example, if points closer than some threshold are regarded as similar to \( p \).
- Use Euclidean distance — restricted to 2D case, where \( p = (p_0, p_1) \) etc.:

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

Anatomy of a Java Function II

```
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);
}
```
Java functions: easy to write your own.

```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);
}
```

Calling a Function

Literal arguments

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

Variable arguments

```java
double p0 = 3.0;
double p1 = 5.0;
double q0 = 14.25;
double q1 = 2.70;
double d = distance(p0, p1, q0, q1);
```

Adding distance() to a Java program

- Take a list of floating points from the command-line.
- Interpret the first pair as \((x, y)\) coordinates of the ‘centre’ point, and every subsequent pair as coordinates of other points.
- Ignore the last argument if length of list is odd
- Calculate the last distance between the ‘centre’ and every other point and print the results

```
% java PointDistance 3 5 6 7 14.25 2.70 1 4.5 2
Distance from (3.0, 5.0) to ( 6.00, 7.00) is 3.61
Distance from (3.0, 5.0) to (14.25, 2.70) is 11.48
Distance from (3.0, 5.0) to ( 1.00, 4.50) is 2.06
```

Flow of Control with Functions

Schematic Structure of Program

```java
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.

Aside: Distance in 3D space

- We did distances for points in 2D space.
- Q: What if we want distances in 3D space or more?
- We can apply the arrays studied last week.
Aside: Distance in 3D space
▶ We did distances for points in 2D space. What if we want
distances in 3D space or more?

public class PointDistance {
public static (double[] p1, double[] p2) {
  double sqd = 0;
  for(int i=0;i<p1.length;i++)
    sqd+=(p1[i]-p2[i])*(p1[i]-p2[i]);
  return Math.sqrt(sqd);
}

public static void main(String[] args) {
double dist1 = distance(new double[] {1.0},
  new double[] {2.0});
double dist2 = distance(new double[] {1.0, 1.0},
  new double[] {0.0, 2.0});
double dist3 = distance(new double[] {1.0, 1.0, 1.0},
  new double[] {1.0, 2.0, 3.0});
}
}

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
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

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`.

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)
```

Cubes, 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

% java Cubes1 6
0 0
1 1
2 8
3 27
4 64
5 125
6 216

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

% 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

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

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, 5

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));
        }
    }
}
```
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.

```
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");
    }
}
```

```
public class Duplication1 {
    public static String joinNames(String n1, String n2){
        return n1 + " "+ n2;
    }

    public static void main(String[] args) {
        String boyName = joinNames("Jock", "McIness");
        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");
    }
}
```

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:

```java
% 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 Static Methods

- Static methods allow you to encode a wide variety of mathematical and data processing functions.
- 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.

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.

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*.