

Sub-Programs and Java Methods

Software Development 1 (F27SA)

Michael Lones

Week 4, lecture 2

Today's Lecture

- What is a sub-program?
- Java methods
- Passing arguments and returning results

What is a sub-program?

A sub-program is a self-contained part of a program that does a particular thing

- It is common practice to split up large programs into a number of smaller sub-programs
- This makes them easier to read and understand
- It also allows the same piece of code to be run multiple times in the same program

Sub-programs in Java

This is the general idea:

```
public class MyProgram {  
    main(String args[]) {  
        // do something  
        // do something else  
    }  
}
```

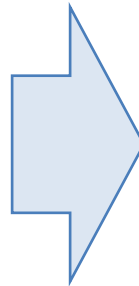


```
public class MyProgram {  
    main(String args[]) {  
        subprogram1();  
        subprogram2();  
    }  
    subprogram1() {  
        // do something  
    }  
    subprogram2() {  
        // do something else  
    }  
}
```

Sub-programs in Java

This is the general idea:

```
public class MyProgram {  
    main(String args[]) {  
        // do something  
        // do something else  
    }  
}
```

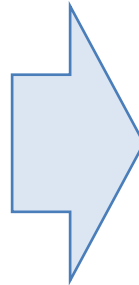


```
public class MyProgram {  
    main(String args[]) {  
        subprogram1();  
        subprogram2();  
    }  
    subprogram1() {  
        // do something  
    }  
    subprogram2() {  
        // do something else  
    }  
}
```

Sub-programs in Java

This is the general idea:

```
public class MyProgram {  
    main(String args[]) {  
        // do something  
        // do something else  
    }  
}
```



```
public class MyProgram {  
    main(String args[]) {  
        subprogram1();  
        subprogram2();  
    }  
    subprogram1() {  
        // do something  
    }  
    subprogram2() {  
        // do something else  
    }  
}
```

Sub-programs in Java

This is the general idea:

```
public class MyProgram {  
    main(String args[]) {  
        // do something  
        // do something else  
    }  
}
```



```
public class MyProgram {  
    main(String args[]) {  
        subprogram1();  
        subprogram2();  
    }  
    subprogram1 () {  
        // do something  
    }  
    subprogram2 () {  
        // do something else  
    }  
}
```

Sub-programs in Java

This is the general idea:

```
public class MyProgram {  
    main(String args[]) {  
        // do something  
        // do something else  
    }  
}
```



```
public class MyProgram {  
    main(String args[]) {  
        subprogram1();  
        subprogram2();  
    }  
    subprogram1 () {  
        // do something  
    }  
    subprogram2 () {  
        // do something else  
    }  
}
```

Red arrows indicate control flow from the main method to the subprograms. One arrow starts at the call to subprogram1() and points to its opening curly brace. Another arrow starts at the call to subprogram2() and points to its opening curly brace. A third arrow starts at the closing curly brace of subprogram1() and points back to the line following the call to subprogram1(). A fourth arrow starts at the closing curly brace of subprogram2() and points back to the line following the call to subprogram2().

Some simple examples

```
public class SubprogramDemo1 {  
    public static void main(String[] args) {  
        for(int i=0; i<=10; i++)  
            System.out.print(i+" ");  
        for(int i=10; i>0; i--)  
            System.out.print(i+" ");  
    }  
}
```

This can be turned into two sub-programs 

Some simple examples

```
public class SubprogramDemo1 {
    public static void main(String[] args) {
        countup();
        countdown();
    }
    static void countup() {          // sub-program 1
        for(int i=1; i<=10; i++)
            System.out.print(i+" ");
    }
    static void countdown() {      // sub-program 2
        for(int i=10; i>0; i--)
            System.out.print(i+" ");
    }
}
```

SubprogramDemo1.java

Some simple examples

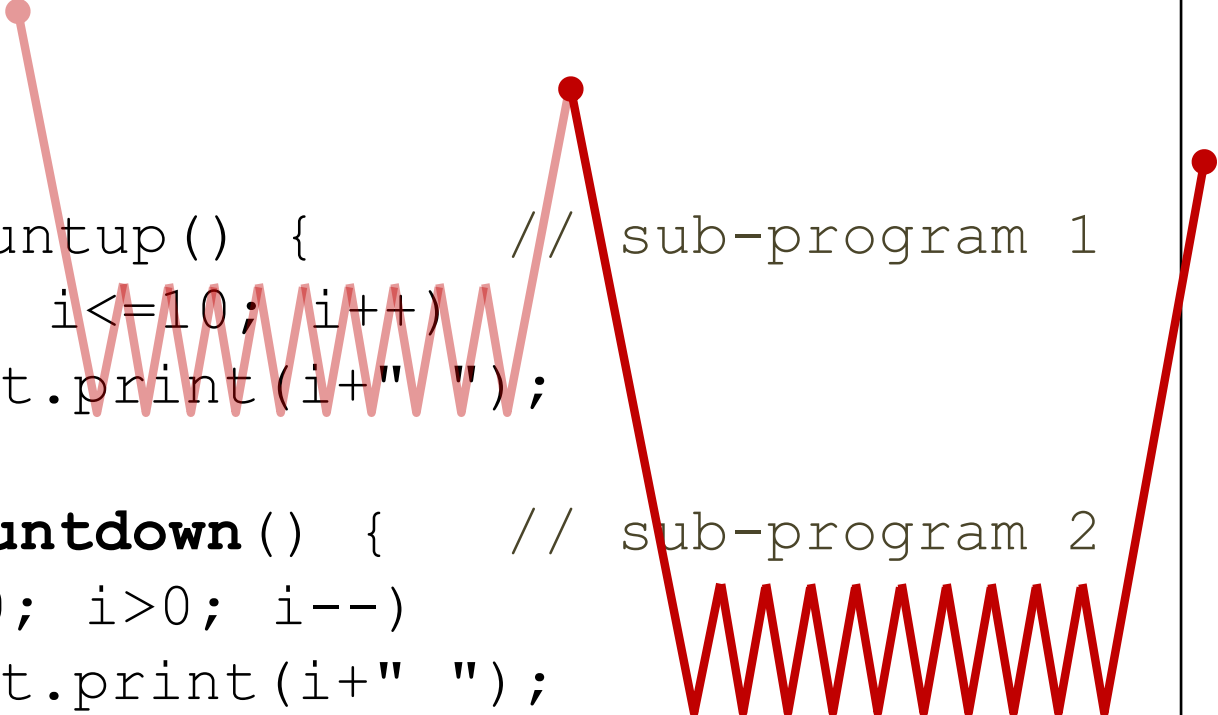
```
public class SubprogramDemo1 {  
    public static void main(String[] args) {  
        countup();  
        countdown();  
    }  
    static void countup() { // sub-program 1  
        for(int i=1; i<=10; i++)  
            System.out.print(i+" ");  
    }  
    static void countdown() { // sub-program 2  
        for(int i=10; i>0; i--)  
            System.out.print(i+" ");  
    }  
}
```

The diagram shows a red zigzag line with dots at the ends and peaks. One dot is at the **countup**() call in the main method. Another dot is at the start of the **countup**() method definition. A third dot is at the start of the **countdown**() method definition. The line zigzags between these three points, illustrating the flow of execution from the main method to the sub-programs.

SubprogramDemo1.java

Some simple examples

```
public class SubprogramDemo1 {  
    public static void main(String[] args) {  
        countup();  
        countdown();  
    }  
    static void countup() {           // sub-program 1  
        for(int i=1; i<=10; i++)  
            System.out.print(i+" ");  
    }  
    static void countdown() {       // sub-program 2  
        for(int i=10; i>0; i--)  
            System.out.print(i+" ");  
    }  
}
```



SubprogramDemo1.java

Some simple examples

We can **call** a sub-program more than once:

```
public class SubprogramDemo2 {  
    public static void main(String[] args) {  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=10; i++)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo2.java

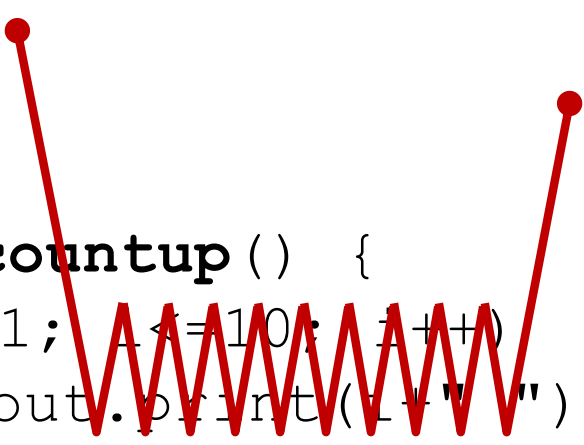
```
$ java SubprogramDemo2  
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
```

Terminal

Some simple examples

We can **call** a sub-program more than once:

```
public class SubprogramDemo2 {  
    public static void main(String[] args) {  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=10; i++)  
            System.out.print(i+" ");  
    }  
}
```



SubprogramDemo2.java

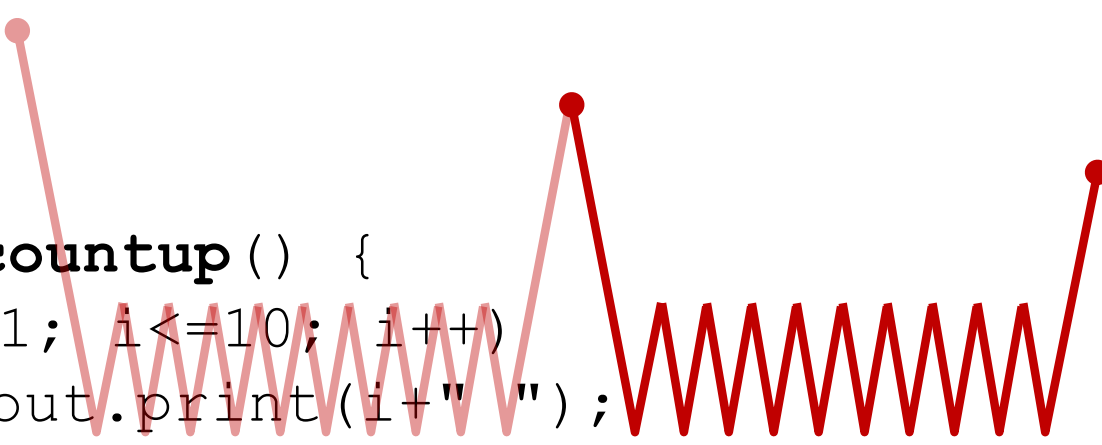
```
$ java SubprogramDemo2
```

```
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
```

Some simple examples

We can **call** a sub-program more than once:

```
public class SubprogramDemo2 {  
    public static void main(String[] args) {  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=10; i++)  
            System.out.print(i+" ");  
    }  
}
```



The diagram illustrates the sequence of calls to the `countup` method. A red line with dots at each call site shows the following sequence: 1. A call from the `main` method to the first `countup` method. 2. A series of 10 recursive calls from `countup` to itself, corresponding to the loop iterations. 3. A return path from the 10th recursive call back to the 9th, and so on, until the first `countup` method returns to the `main` method. The line is light red for the first call and dark red for the subsequent recursive calls.

```
$ java SubprogramDemo2
```

```
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
```

Any Questions?

Some simple examples

What if a variable is used in both parts of the program?

```
public class SubprogramDemo3 {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
  
        for(int i=1; i<=max; i++) {  
            Sytem.out.print(i+" ");  
        }  
  
        for(int i=1; i<=max; i++) {  
            Sytem.out.print(i+" ");  
        }  
    }  
}
```

5

1 2 3 4 5 1 2 3 4 5

Terminal

Some simple examples

Can we still do this?

```
public class SubprogramDemo3a {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=max; i++)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo3a.java

Some simple examples

Can we still do this?

```
public class SubprogramDemo3a {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=max; i++)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo3a.java

No, this won't work.
The code in a sub-program can not see any of the code in `main`, including its variables.

Scope

A variable can only be seen, used or modified within the block where it is declared

- This is known as a variable's **scope**
- It means that a variable declared in one sub-program can not be seen in another sub-program

Scope

A variable can only be seen, used or modified within the block where it is declared

- This is known as a variable's **scope**
- It means that a variable declared in one sub-program can not be seen in another sub-program
- More generally, a variable declared within {...}s can not be seen outside of the {...}s.
- This includes variables declared in **for** loops, where the **for** statement is considered part of the block

Scope

A variable can only be seen, used or modified within the block where it is declared

So, this piece of code won't work:

```
for (int i=1; i<=10; i++) {  
    System.out.println(i);  
}  
System.out.println("Looped "+i+ " times.");
```

Since **i** can only be seen in the for loop's block, and not in the code following the for loop.

Information Hiding

Scope is an example of a broader programming concept known as **information hiding**

- Basically, each part of a program should know as little as possible about other parts of the program
- This means different parts of the program are less likely to interfere with each other
- It also means that different parts of a program can be written separately, potentially by different people

Some simple examples

So, how do we get this working?

```
public class SubprogramDemo3a {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup();  
        countup();  
    }  
    static void countup() {  
        for(int i=1; i<=max; i++)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo3a.java

Passing arguments

We can **pass an argument** to the sub-program:

```
public class SubprogramDemo3b {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup(max);  
        countup(max);  
    }  
    static void countup(int to) {  
        for(int i=1; i<=to; i++)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo3b.java

Passing arguments

We can **pass an argument** to the sub-program:

```
public class SubprogramDemo3b {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup(max);  
        countup(max);  
    }  
    static void countup(int to) {  
        for(int i=1; i<=to; i++)  
            System.out.print(i+" ");  
    }  
}
```

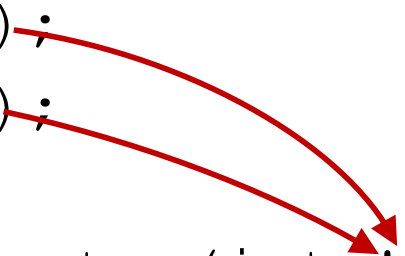
SubprogramDemo3b.java

The value of **max** gets copied into a new variable **to** each time countup is called

Passing arguments

We can **pass an argument** to the sub-program:

```
public class SubprogramDemo3b {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int max = scan.nextInt();  
        countup(max);  
        countup(max);  
    }  
    static void countup(int to) {  
        for(int i=1; i<=to; i++)  
            System.out.print(i+" ");  
    }  
}
```

Two red curved arrows originate from the **max** parameter in the two `countup(max)` calls within the `main` method. Both arrows point to the **to** parameter in the `countup` method signature, illustrating how the value of `max` is passed as an argument to `countup`.

SubprogramDemo3b.java

So, we are explicitly indicating which information to share with the sub-program

Passing arguments

We can **pass an argument** to the sub-program:

```
public class SubprogramDemo3c {  
    public static void main(String[] args) {  
        countup(2);  
        countup(4);  
        countup(6);  
        countup(8);  
    }  
}
```

SubprogramDemo3c.java

We can call the sub-program multiple times with different arguments.

```
$ java SubprogramDemo3c  
1 2 1 2 3 4 1 2 3 4 5 6 1 2 3 4 5 6 7 8
```

Terminal

Passing arguments

We can **pass multiple arguments** to a sub-program:

```
public class SubprogramDemo4 {  
    public static void main(String[] args) {  
        count(5, 43, 3);  
    }  
    static void count(int from, int to, int step) {  
        for(int i=from; i<=to; i+=step)  
            System.out.print(i+" ");  
    }  
}
```

SubprogramDemo4.java

```
$ java SubprogramDemo4  
5 8 11 14 17 20 23 26 29 32 35 38 41
```

Terminal

Any Questions?

Returning values

How do we get values back from a sub-program?

```
public class ReturnValueDemo1 {  
    public static void main(String[] args) {  
        double value = 5;  
        square(value);  
        System.out.println(value);  
    }  
    static void square(double arg) {  
        arg = arg * arg;  
    }  
}
```

Not like this! If you change the value of an argument, its value won't get updated in the calling code. Passing arguments is a one-way process (at least for primitives...)

```
$ java SubprogramDemo1  
5
```

Terminal

Returning values

A sub-program can explicitly **return** a value:

```
public class ReturnValueDemo1 {  
    public static void main(String[] args) {  
        double y = square(5);  
        System.out.println(y);  
    }  
    static double square(double arg) {  
        double result = arg * arg;  
        return result;  
    }  
}
```

ReturnValueDemo1.java

```
$ java SubprogramDemo1  
25
```

Terminal

Returning values

A sub-program can explicitly **return** a value:

```
public class ReturnValueDemo1 {  
    public static void main(String[] args) {  
        double y = square(5);  
        System.out.println(y);  
    }  
    static double square(double arg) {  
        double result = arg * arg;  
        return result;  
    }  
}
```

ReturnValueDemo1.java

We replace **void** with the **return** type, and use the **return** keyword to return a value

```
$ java SubprogramDemo1  
25
```

Terminal

Returning values

A sub-program can explicitly **return** a value:

<pre>public class ReturnValueDemo1 { public static void main(String[] args) { double y = square(5); System.out.println(y); } static double square(double arg) { double result = arg * arg; return result; } }</pre>	ReturnValueDemo1.java
--	-----------------------

The **method call** then **evaluates to** this return value

We replace **void** with the **return type**, and use the **return** keyword to return a value

```
$ java SubprogramDemo1  
25
```

Terminal

Returning values

A sub-program can contain **multiple return statements**:

```
public class ReturnValueDemo2 {  
    public static void main(String[] args) {  
        System.out.println( max(2, 4) );  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```

ReturnValueDemo2.java

```
$ java SubprogramDemo2  
4
```

Terminal

Returning values

Return values can be used in any expression:

```
public class ReturnValueDemo3 {  
    public static void main(String[] args) {  
        System.out.println( square(5) * max(2, 4) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;;  
        else return arg2;  
    }  
}
```

ReturnValueDemo2.java

Returning values

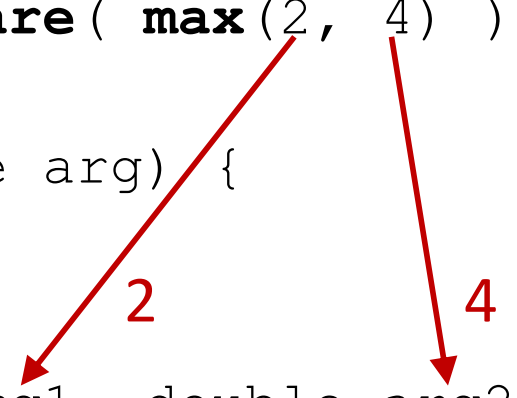
Return values can be used as arguments:

```
public class ReturnValueDemo4 {  
    public static void main(String[] args) {  
        System.out.println( square( max(2, 4) ) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```

Returning values

Return values can be used as arguments:

```
public class ReturnValueDemo4 {  
    public static void main(String[] args) {  
        System.out.println( square( max(2, 4) ) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```



The diagram illustrates the flow of arguments in the provided Java code. Two red arrows originate from the arguments '2' and '4' within the `max` method call in the `main` method. The first arrow, labeled with a red '2', points to the `arg1` parameter of the `max` method. The second arrow, labeled with a red '4', points to the `arg2` parameter of the `max` method. This visualizes how the values 2 and 4 are passed as arguments to the `max` method, which then returns its result to the `square` method.

Returning values

Return values can be used as arguments:

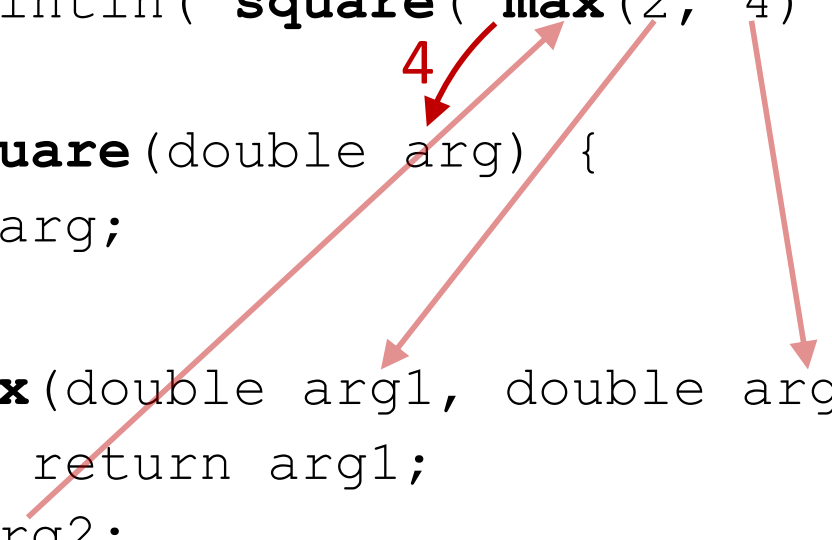
```
public class ReturnValueDemo4 {  
    public static void main(String[] args) {  
        System.out.println( square( max(2, 4) ) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```

4

Returning values

Return values can be used as arguments:

```
public class ReturnValueDemo4 {  
    public static void main(String[] args) {  
        System.out.println( square( max(2, 4) ) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```



The diagram illustrates the flow of return values in the provided Java code. It features three red arrows: one originates from the number '4' in the `max` function call within the `main` method and points to the `max` function definition; a second originates from the `max` function definition and points to the `square` function call; and a third originates from the `square` function call and points to the `System.out.println` statement. Additionally, a red number '4' is placed above the `max` function definition, indicating its return value.

Returning values

Return values can be used as arguments:

```
public class ReturnValueDemo4 {  
    public static void main(String[] args) {  
        System.out.println( square( max(2, 4) ) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double max(double arg1, double arg2) {  
        if(arg1>arg2) return arg1;  
        else return arg2;  
    }  
}
```

16

Any Questions?

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

11.180339887498949

Terminal

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

The diagram illustrates the flow of arguments from the `main` method to the `pythagoras` method. Two red arrows originate from the arguments `5` and `10` in the `pythagoras` call within the `main` method. The first arrow points to the parameter `a` in the `pythagoras` method signature, with the number `5` written in red above the arrow. The second arrow points to the parameter `b` in the `pythagoras` method signature, with the number `10` written in red above the arrow.

11.180339887498949

Terminal

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

The diagram illustrates the recursive nature of the code. A red arrow points from the **pythagoras** method call in the **main** method to the **pythagoras** method definition. Another red arrow points from the **pythagoras** method definition to the **square** method definition. A third red arrow points from the **square** method definition to the **square** method calls within the **pythagoras** method definition. The number 5 is written in red next to the first arrow.

11.180339887498949

Terminal

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

The diagram illustrates the recursive calls between the `pythagoras` and `square` methods. Red arrows show the following sequence of calls:

- From `pythagoras(5, 10)` in the `main` method to `square(5)` in the `pythagoras` method.
- From `square(5)` back to `pythagoras` (the first call).
- From `pythagoras` to `square(10)` in the `pythagoras` method.
- From `square(10)` back to `pythagoras` (the second call).
- Finally, from the second `pythagoras` call back to the `main` method.

The value **25** is shown in red, indicating the result of the first `square(5)` call.

11.180339887498949

Terminal

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

The diagram illustrates the recursive nature of the code. Red arrows show the following call sequence: 1. The `main` method calls `pythagoras(5, 10)`. 2. The `pythagoras` method calls `square(a)` (where `a` is 5). 3. The `pythagoras` method also calls `square(b)` (where `b` is 10). 4. The `square` method is called twice, once for each argument. A red number '10' is placed near the second `square` call to highlight the argument value.

11.180339887498949

Terminal

Sub-programs calling sub-programs

It's common for sub-programs to call one another:

```
public class CalculateHypotenuse {  
    public static void main(String[] args) {  
        System.out.println( pythagoras(5, 10) );  
    }  
    static double square(double arg) {  
        return arg * arg;  
    }  
    static double pythagoras(double a, double b) {  
        double val = square(a) + square(b);  
        return Math.sqrt(val);  
    }  
}
```

The diagram illustrates the recursive calls between the `pythagoras` and `square` methods. Red arrows show the flow of execution: from the `main` method to `pythagoras(5, 10)`, then to `square(5)` and `square(10)`, and finally back to `pythagoras` to calculate the final result. A red number '100' is placed near the `square` call in the `pythagoras` method, indicating the value returned by `square(10)`.

11.180339887498949

Terminal

Terminal

Returning multiple values

Is it possible for a sub-program to return multiple values?

- Not in most programming languages, Java included
- Some languages (e.g. Go) do support this
- It can be achieved in Java using **data structures**
- More on this in the next lecture

Method signatures

The correct name for a sub-program in Java is a **method**. Every method has a **signature**.

A method's signature indicates:

- the name that is used to call it
- the arguments it expects
- the type of the value it returns
- other things, such as `public` and `static`, which will be explained later in SD1

Method signatures

The correct name for a sub-program in Java is a **method**. Every method has a **signature**, e.g.

```
int countLetters(String s)
```

```
String join(String s1, String s2)
```

```
double calculateVolume(double radius)
```

```
void outputMessage(String message)
```



void means that the method
does not return anything

Any Questions?

Quiz

Will this work?

```
public class Q1 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 5;  
        add();  
    }  
    static void add() {  
        System.out.println(a+" "+b+"="+ (a+b) );  
    }  
}
```

Quiz

Will this work?

```
public class Q2 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 5;  
        add(a, b);  
    }  
    static void add(int a, int b) {  
        System.out.println(a+" "+b+"="+ (a+b) );  
    }  
}
```

Quiz

Will this work?

```
public class Q3 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 5;  
        add(a, b);  
    }  
    static int add(int a, int b) {  
        System.out.println(a+" "+b+"="+ (a+b) );  
    }  
}
```


Quiz

Will this work?

```
public class Q4 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 5;  
        c = add(a, b);  
        System.out.println(a+" "+b+"="c) ;  
    }  
    static void add(int a, int b) {  
        return a+b;  
    }  
}
```

Quiz

Will this work?

```
public class Q5 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 5;  
        int c = divide(a, b);  
        System.out.println(a+"/"+b+"="c);  
    }  
    static double divide(double a, double b) {  
        return a / b;  
    }  
}
```

Multiplication Tables Revisited

```
public class MultiplicationTables {  
    public static void main(String[] args) {  
        // variable declarations  
        int maxtable; // largest multiplicand  
        int maxvalue; // largest multiplier  
        int product;  // multiplier x multiplicand  
  
        // get input from user  
        Scanner scan = new Scanner(System.in);  
        System.out.println("What number would you like  
                             to produce tables up to?");  
        maxtable = scan.nextInt();  
        System.out.println("What is the maximum  
                             multiplier for each table?");  
        maxvalue = scan.nextInt();  
    }  
}
```

MultiplicationTables.java

Multiplication Tables Revisited

```
// output tables
for(int table=1; table<=maxtable; table++) {
    System.out.println(
        "Multiplication table for "+table);
    for(int value=1; value<=maxvalue; value++) {
        product = table * value;
        System.out.println(
            table+" x "+value+" = "+product);
    }
}
}
```

Remember this program from the iteration lecture? Let's move the table printing code into a separate method:

Multiplication Tables Revisited

```
// output tables
for(int table=1; table<=maxtable; table++) {
    outputTable(table, maxvalue);
}

// output a single multiplication table
static void outputTable(int table, int upto) {
    int product; // multiplier x multiplicand
    System.out.println(
        "Multiplication table for "+table);
    for(int value=1; value<=upto; value++) {
        product = table * value;
        System.out.println(table+" x "+value+" = "+product);
    }
}
```

Now move all the calculation code into a separate method:

Multiplication Tables Revisited

```
        outputTables(maxtable, maxvalue);
    }

    // output multiplication tables
    static void outputTables(int uptotable, int uptovalue) {
        for(int table=1; table<=uptotable; table++) {
            outputTable(table, uptovalue);
        }
    }

    // output a single multiplication table
    static void outputTable(int table, int upto) {
        int product; // multiplier x multiplicand
        System.out.println("Multiplication table for "+table);
        for(int value=1; value<=upto; value++) {
            product = table * value;
            System.out.println(table+" x "+value+" = "+product);
        }
    }
}
```

Multiplication Tables Revisited

```
        outputTables(maxtable, maxvalue);
    }

    // output multiplication tables
    static void outputTables(int uptotable, int uptovalue) {
        for(int table=1; table<=uptotable; table++) {
            outputTable(table, uptovalue);
        }
    }

    // output a single multiplication table
    static void outputTable(int table, int upto) {
        int product; // multiplier x multiplicand
        System.out.println("Multiplication table for "+table);
        for(int value=1; value<=upto; value++) {
            product = table * value;
            System.out.println(table+" x "+value+" = "+product);
        }
    }
}
```

Note that any method can call any other method!

Multiplication Tables Revisited

Now let's add to the behaviour of the program:

```
public class ArithmeticTables {  
    public static void main(String[] args) {  
        // variable declarations  
        int maxtable; // largest table  
        int maxvalue; // largest argument  
  
        // get input from user  
        Scanner scan = new Scanner(System.in);  
        System.out.println("What number would you like  
                           to produce tables up to?");  
        maxtable = scan.nextInt();  
        System.out.println("What is the maximum  
                           argument for each table?");  
        maxvalue = scan.nextInt();  
  
        outputTables(maxtable, maxvalue); // print out tables
```

ArithmeticTables.java

Multiplication Tables Revisited

```
// output arithmetic tables
static void outputTables(int uptotable, int uptovalue) {
    for(int table=1; table<=uptotable; table++) {
        outputMultTable(table, uptovalue);
        outputDivisionTable(table, uptovalue);
    }
}

// output a single multiplication table
static void outputMultTable(int table, int upto) { ... }

// output a single division table
static void outputDivisionTable(int table, int upto) {
    double quotient; // dividend / divisor
    System.out.println("Division table for "+table);
    for(int value=1; value<=upto; value++) {
        quotient = table / (double) value;
        System.out.println(table+" / "+value+" = "+product);
    }
}
}
```

Multiplication Tables Revisited

Terminal

```
$ java ArithmeticTables
What number would you like to produce tables up to?
3
What is the maximum argument for each table?
3
Multiplication table for 1
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
Division table for 1
1 / 1 = 1
1 / 2 = 0.5
1 / 3 = 0.3333333333333333

Multiplication table for 2
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
```

Multiplication Tables Revisited

What does this example demonstrate?

- Methods can break up code into manageable chunks, making a program more readable and maintainable
- They make it easier to add new functionality, without disturbing the functionality that's already there
- They are also useful for separating **interface** code from **implementation** code (more on this in SD2)
- In short, they are an important aspect of software development

Next Lecture

- Multi-dimensional arrays
- Passing arrays to/from methods

Tutorial 3 and Lab 4

- Array and Method exercises

Summary

- Sub-programs are used to break up a program into smaller chunks, and allow code to be reused
- Sub-programs are known as **methods** in Java
- Methods can be passed one or more arguments
- Methods can return exactly one value
- The **method signature** specifies the arguments and the return type