Writing methods

Many people quit looking for work when they find a job: hahahahahaha!!

What do you think????

Lecture objectives

To be able to understand the following fundamental concepts of the Java programming language:

- Methods
- Data scope
- Debugging

Classes and methods

- To date every class has had one method - `main`
  - It is the starting point for an application
  - These classes are called *driver* programs

- We have used (invoked) prewritten methods
  - eg from the Math class: `pow(),  random()`
  - eg from the String class: `toUpperCase(), charAt()`

- We know how they work but we didn't write them
- We also need to write general purpose methods

What are methods?

- Methods *do* things
- They are like an independent mini-program
- By giving a name to a series of statements a program is easier to read and debug
- In some programming languages they are called
  - functions, procedures, subprograms, subroutines
- It is good programming style to provide methods for doing particular functions
  - ie isolate the function in a method

How Methods Work

- Sequence of events when a method is called:
  - the program “jumps” to that method.
  - the arguments in the call (if any) are copied into the method’s corresponding parameters.
  - the method begins executing.
  - when the method is finished, the program “returns” to the point at which the method was called.
  - If there is a `return` statement, the value specified is returned also

A `main` method

```
public class Welcome
{
    public static void main(String[] args)
    {
          System.out.println(“Ladies and gentlemen – welcome to the Snakepit”);
          System.out.println(“And we hope you enjoy the show”);
    }
}
```

- If the first print line is to be reused, we could create another method (also *static* as called from the *static* method `main`, not by an object)

A new Method with no parameters

```
public class Welcome
{
    public static void main(String[] args)
    {
          welcomeMessage();
          System.out.println(“And we hope you enjoy the show”);
    }

    public static void welcomeMessage()
    {
          System.out.println(“Ladies and gentlemen – welcome to the Snakepit”);
    }
}
```

Like a telephone call

main() is the calling method

welcomeMessage() is the called or invoked method

- As the static called method is located in the same class
  - Only the method name is required to call the method

```
welcomeMessage()
```

- If the static called method is located in another class
  - The method name must be preceded by the class name and period

```
Math.pow(2,2)
```

Methods with parameters

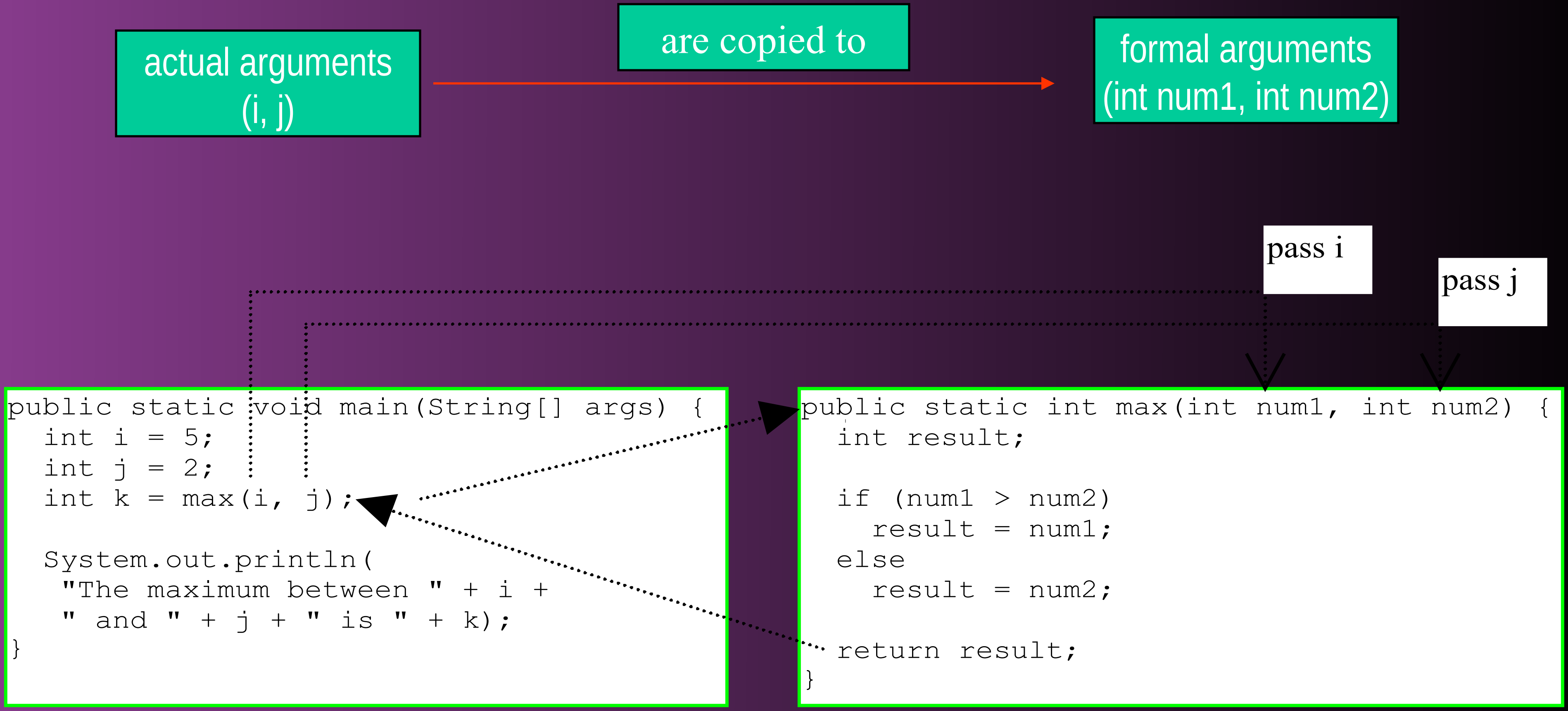
- Recall the Math `max` method we have studied eg.

  `static int max(int num1, int num2)`

- We could also write a `max` method

```
public static  int  max (int num1, int num2)
{
     int result;
     if (num1 > num2)
               result = num1;
     else
               result =  num2;

    return result;
}
```

How it works



actual arguments
(i, j)

are copied to

formal arguments
(int num1, int num2)

pass i

pass j

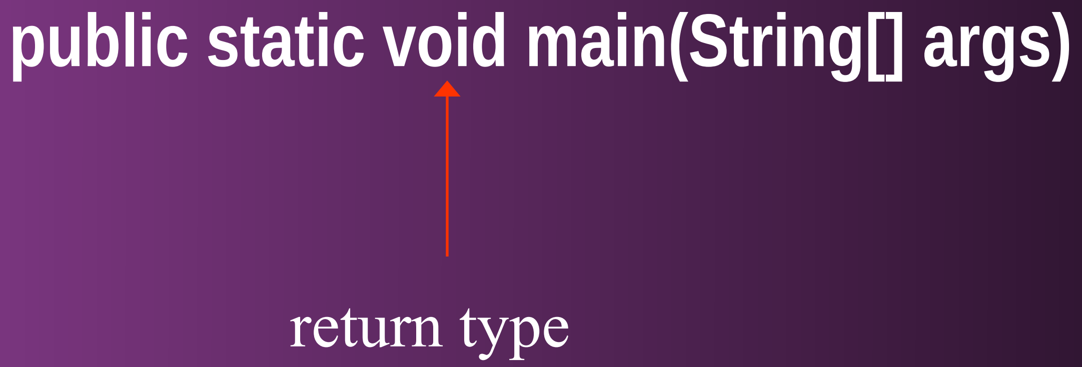
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

return type must match receiving type

Void methods

- `void` methods do not return a value
- They do not have a `return` statement
  - their return type is stated as `void`
- eg `main` method



```
public static void main(String[] args)
```

return type

Passing one parameter

- Write a main method that
  - prompts the user for their name
  - calls a method `displayString` and passes the name to it

- Write a method `displayString` that
  - accepts a String parameter and displays it

```
import javax.swing.*;
public class TestVoid
{
    public static void main(String[] args)
    {
            String name = JOptionPane.showInputDialog ("Enter your name");
           displayString(name);      // the static method displayString is defined in the SAME class
    }                                 // so no class name is required to invoke it

    public static void displayString (String str)
    {
           System.out.println(str);
    }
}
```

More practice

- Write a main method that
  - prompts a user to enter an integer
  - passes that integer to a method called cube
  - receives the cube of that integer back from the method
  - displays the return value

- Write a method called cube that
  - accepts an integer, cubes it and returns the result

```
import javax.swing.*;
public class TestCube
{
    public static void main(String[] args)
    {
        String numStr = JOptionPane.showInputDialog (“Enter an integer to cube: “);
        int num = Integer.parseInt(numStr);
        int cubed = cube (num);
        System.out.println(num + “\t” + cubed);
    }

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

More practice

- Change the preceding code so the `main` method has a *for* loop
- It passes the numbers 0 to 3 to the `cube` method, one at a time
- The `cube` method is unchanged
- This is an example of reuse of a method

```
public class TestCube
{
    public static void main(String[] args)
    {
          for (int num = 0; num < 4; num++)
          {
               int cubed = cube (num);
               System.out.println(num + “\t” + cubed);
          }
    }

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

```
0    0

1    1

2    8

3    27
```

Writing it differently…

```
public class TestCube
{
    public static void main(String[] args)
    {
        for (int num = 0; num < 4, num++)
        {
            System.out.println (num + “\t” + cube(num));     //prints the result of a method call
        }
    }

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

Passing two Parameters

- Write a `main` method that
  - prompts a user for a message to print out
  - prompts a user for the number of times it is to be printed
  - passes those two inputs to a method called `doPrint`

- Write a void method called `doPrint` that
  - receives the above 2 parameters
  - uses a for loop to control the number of times the message is printed

```java
import javax.swing.*;
public class TestCube
{
    public static void main(String[] args)
    {
        String messageText = JOptionPane.showInputDialog(“Enter a message to print out: “);
        String numStr = JOptionPane.showInputDialog(“Print it how many times?: “);
        int num = Integer.parseInt(numStr);
       doPrint(messageText, num);
    }


    public static void doPrint(String message, int n)
    {
           for (int i = 0; i < n; i++)
               System.out.println(message);
    }
}
```

Boolean methods

- Return a `boolean` type ie true or false
- Name the method so that it makes sense as true or false

```
import javax.swing.*;
public class OddEven
{
    public static void main(String[] args)
    {
        String numStr = JOptionPane.showInputDialog("Enter an integer: ");
        int number = Integer.parseInt(numStr);

        if (isEven(number))
            System.out.println(number + " is an even number");
        else
            System.out.println(number + " is an odd number");
    }

    public static boolean isEven (int num)
    {
        if (num%2 == 0)
            return true;
        else
            return false;
    }
}
```

Overloading Methods

- *Method overloading* is using the same method name for multiple methods
- The header of each overloaded method must be unique
- The header includes
  - the return type,
  - the order and the number of parameters

```
public static double addNumbers (double a, double b)
public static double addNumbers (double a, double b, double c)
```

- The compiler must be able to determine which version of the method is being invoked by analysing the signature

```
public class MethodOverloadTest
{
    public static void main (String [] args)
    {
          double result = addNumbers (25.0, 4.32);
          System.out.out.println(result);
    }

    public static double addNumbers (double a, double b)
    {
          return a + b;
    }

    public static double addNumbers (double a, double b, double c)
    {
          return a + b + c;
    }
}
```

Overloaded Methods

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

etc.

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:“ + total);
System.out.println (total);
```

Overloaded transport method

Data scope

- The *scope* or *visibility* of a variable (or constant) is the area in a program in which that data can be used (referenced)

- *Local* variables and *Parameter* variables
  - are declared within a method and can only be used in that method.
  - They are created on the line where they are declared and destroyed when the method is exited

Variable scope

```
public static char calc (int num1, int num2, String message)
 {
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
 }
```

num1, num2 and message are *parameter variables*

sum and result are *local variables*

They are created each time the method is called, and are destroyed when it finishes executing

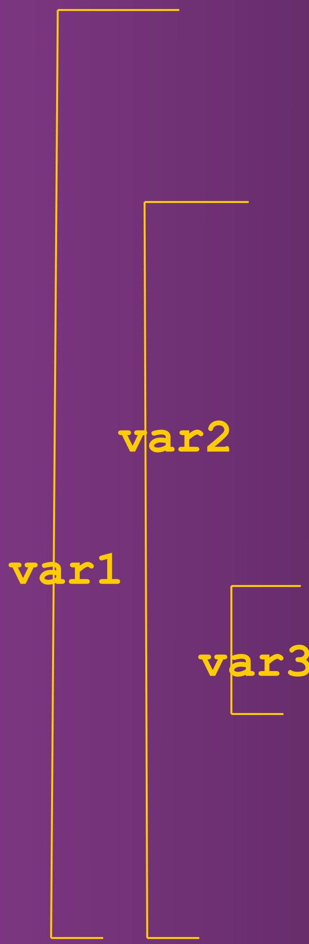
Blocks and scope

- Blocks can be separate or nested but never overlapping

```
{

    {

    }

}
```

- When you declare a variable within a block you cannot access it outside the block – it ceases to exist
  - A nested block is within the scope of an outer block
- The portion of a program where you can reference a variable is its *scope*

Variable Scope



var2

var1

var3

```
public static void aMethod (int var1)
{
   ...
   int var2;

   ...
   if (var1 > var2)
   {
      ...
      int var3;
      ...
    }

    if (var3 < var1) //wont compile
       ...;
}
```

Scope of Local Variables, cont.

- The scope of a local variable declared in a `for` loop header is the `for` loop block

```
public static void correctMethod()
{
    int x = 1, y = 1;

    for (int i = 1; i < 10; i++)        // i is declared
    {
        x = x + i;
    }

    for (int i = 1; i < 10; i++)       // i is declared again
    {
        y = y + i;
    }
}
```

A common mistake

- ***Declaring a variable in a block and then trying to use it outside the block***

```
for (int i = 1; i < 10; i++)
{
   x = x + i;
}

System.out.println(“the value of  i = “ +  i);
```

- This will cause a compilation error

Parameters and scope

- We pass parameters to methods because of scope
- The parameter `num` is passed to the `cube` method because the `cube` method cannot access it – it is out of scope

```
public class TestCube
{
    public static void main(String[] args)
    {
        for (int num = 0; num < 6; num++)
            System.out.println(num + “\t” + cube(num);
    }

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

BlueJ Debugger

- Demonstrate BlueJ debugger (eg *OddEven*)
- A debugger is an essential tool for finding logic errors
- What functions does it provide?
  - Setting breakpoints
    - This stops program execution at this point and displays the code
    - Click in the area to the left of the text in the text editor
  - Stepping through the code
    - *Step* line by line
    - *Step into* a method
  - Inspecting variables
    - These are automatically displayed

Assessment 2

- As per the assessment schedule, the assessment programs from weeks 4 and 5 are due in your workshop today
- You must use the Workshop Assessment Submission Template to submit your work. It is in the “Workshop documents” on *eCourse*.
- It must be handed to your tutor at the commencement of the workshop
- you must be present and be ready to demonstrate and answer questions about the programs

Lecture Outcomes

Today we have covered:

- Methods
- Data scope
- Debugging

- Questions?