Methods and Parameters (JFS Chap 5)

- In this Section:
  - Use of *methods* to divide a program into more manageable chunks
  - Using library methods, and designing our own
  - The use of *parameters*
  - Building on new methods to give more methods...

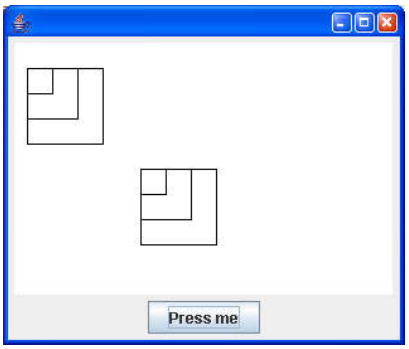
Why Methods?

- Real programs can be quite long and complex
- The main solution to this problem:
  - Split the program into sections known as "methods"
  - "Divide and conquer" – tackle them more or less independently
- If well chosen and designed, methods are good "building blocks"
  - Reusable in other applications

The basic idea of methods: The LogoMethod example (JFS, p61)

- Suppose that we want an application to draw two simple logos:

Demo
LogoMethod



- Each logo comprises:
  - Three squares (rectangles)
  - With the same top left corner position
  - With side lengths 20, 40 and 60 pixels

- To draw the logos we could simply use six `drawRect`s in `actionPerformed`:

```
paper.drawRect(10, 20, 60, 60);
paper.drawRect(10, 20, 40, 40);
paper.drawRect(10, 20, 20, 20);
paper.drawRect(100, 100, 60, 60);
paper.drawRect(100, 100, 40, 40);
paper.drawRect(100, 100, 20, 20);
```

- When designing a program, we should always think about the future:
  - "Maybe tomorrow I'll need to draw the logos in different positions"
  - This kind of change is eased if we introduce *some new variables* to hold some of the basic data values...

- With some new variables, `xPos` and `yPos`, we could have:

```
xPos = 10;
yPos = 20;
paper.drawRect(xPos, yPos, 60, 60);
paper.drawRect(xPos, yPos, 40, 40);
paper.drawRect(xPos, yPos, 20, 20);
xPos = 100;
yPos = 100;
paper.drawRect(xPos, yPos, 60, 60);
paper.drawRect(xPos, yPos, 40, 40);
paper.drawRect(xPos, yPos, 20, 20);
```

- The logos are easily moved – limited changes to assignments only
- But:
  - It is a little irritating having duplicated sections of code
  - And suppose that we need to draw *many more* logos...

Methods

- What if we want to draw several logos?
  - We could duplicate all the `drawRect`s ...
  - ... it would get *very messy and unmanageable*!
- The solution is to introduce a *new* ***method*** that draws a *single logo*
  - We bundle all the logo-drawing code into a separate named section of the program (a method)
  - Then whenever we wish to draw a logo, we simply "*call*" *the method* by mentioning its name in a special method call statement
  - Java carries out a method call statement by temporarily diverting its sequence of execution via the named method
- We have already used calls of library methods in exactly this way – now we will write our own

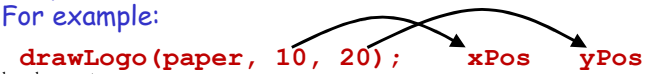
Parameters

- But different logos must be drawn at different positions
  - On slide 5 we *assigned* variables `xPos` and `yPos` appropriate values
- Now, each time we call our method, we must be able to specify the appropriate values for `xPos` and `yPos` :
  - This is what **parameters** are for
- Our method `drawLogo` will have two parameters standing for:
  - The top left corner x and y coordinates
- (It will also need a parameter standing for the "paper" on which the logo will be displayed)

The new method

```
1.   private void drawLogo(Graphics drawingArea,
2.                         int xPos, int yPos) {
3.       drawingArea.drawRect(xPos, yPos, 60, 60);
4.       drawingArea.drawRect(xPos, yPos, 40, 40);
5.       drawingArea.drawRect(xPos, yPos, 20, 20);
6.   }
```

- The core of this is the three statements that draw a single logo
  - In terms of the variables `xPos` and `yPos`
- `xPos` and `yPos` are *not* assigned values here
  - When writing the method, we cannot know what the values should be!
  - They will receive their values *from each call of the method* For example:



Key points

- The *method header* (lines 1, 2) of `drawLogo`
  - States `private void` (more about this later) and the **method name**
  - Gives *names* for the parameters so they can be referred to within the **body** of the method
    (These names are called the "**formal** parameters")
  - Fixes the *order* and *types* of the parameters
- The **body** of the method (lines 3, 4, 5), between `{ }`
  - Specifies the action to be taken when the method is called
  - *Refers to the parameter names where needed*
- The parameter names are used *within the method body* to stand for *places where the specific values from each method call should be used*:

```
drawingArea.drawRect(xPos, yPos, 60, 60);
```


The new method in context

```
imports ... main ... createGUI ...

    public void actionPerformed(ActionEvent event) {
        Graphics paper = panel.getGraphics();
1.      drawLogo(paper, 10, 20);
2.      drawLogo(paper, 100, 100);
    }

    private void drawLogo(Graphics drawingArea,
                          int xPos, int yPos) {
3.      drawingArea.drawRect(xPos, yPos, 60, 60);
4.      drawingArea.drawRect(xPos, yPos, 40, 40);
5.      drawingArea.drawRect(xPos, yPos, 20, 20);
    }
```

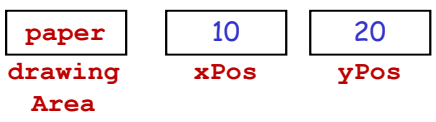
Key points

- In addition to `main` and `createGUI`, the program contains *two* methods: `actionPerformed` and `drawLogo`
  - The order that the methods appear does not matter to Java
  - But they *must* be separate – *not enclosing nor overlapping*
  - The order in which they are *called* determines when their actions are carried out
- The effect of the `actionPerformed` method (when it is called) is to draw two logos via two further method calls
- `actionPerformed` is a "`public`" method:
  - It is called from "elsewhere" - by the JVM
- `drawLogo` is a "`private`" method
  - It may be called only from *within* this proram

- Note: `drawLogo` *must* have the `Graphics` parameter
  - so `actionPerformed` can tell it what to draw on!
- Lines 1 and 2 are calls of `drawLogo`
- Whenever we call a method:
  - The values of the "**actual** parameters" (here the actual numbers) are transferred ("passed") to the method
  - The values are copied into *new* variables which have the formal parameter names
  - For example: with the call

    ```
    drawLogo(paper, 10, 20);
    ```

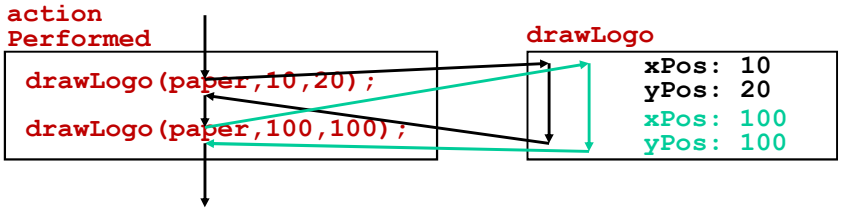
    the formal parameters are set up as *new variables* like this:



Then the call really starts:

- Execution skips from the "point of call" (in `actionPerformed`) *to the start of the* `drawLogo` *method body* (line 3) …
- … runs through lines 3-5, carrying out the statements …
- … then ***returns*** to the *point of call* and continues from there to the next step in `actionPerformed`

- The dynamic picture:



"Local" Variables

- We have seen how we can declare and use variables inside `actionPerformed`
  - In fact, *any* method can declare such "local" variables *for its own private use*
- To illustrate this, let's reconsider the `drawLogo` method. It contains these lines:

```
drawingArea.drawRect(xPos, yPos, 60, 60);
drawingArea.drawRect(xPos, yPos, 40, 40);
drawingArea.drawRect(xPos, yPos, 20, 20);
```

- If we wished to alter the program to draw a different size or differently proportioned logo, then we could (painstakingly) edit all the numbers
- Or, as with the earlier example it would be easier to introduce some extra variables… (see next slide)

```
private void drawLogo(Graphics drawingArea,
                      int xPos, int yPos) {
    int step = 20;
    int smallest = 20;
    int middle = smallest + step;
    int largest = middle + step;
    drawingArea.drawRect(xPos,yPos,largest,largest);
    drawingArea.drawRect(xPos,yPos,middle,middle);
    drawingArea.drawRect(xPos,yPos,smallest,smallest);
}
```

Not in JFS

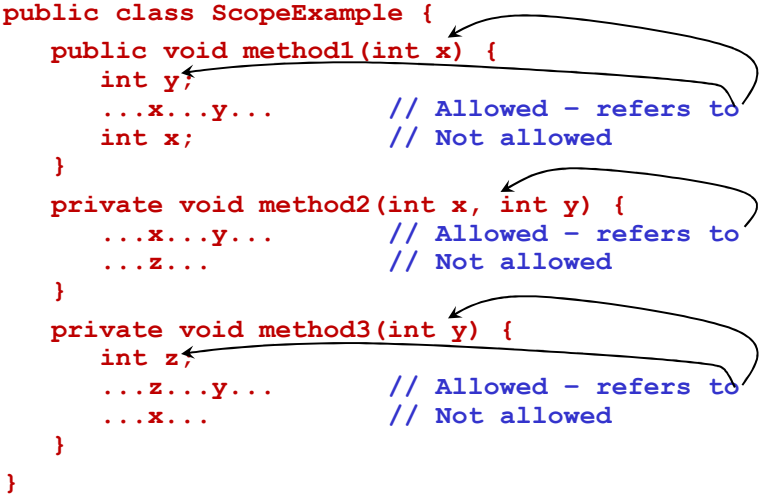
- This version of `drawLogo` is exactly equivalent in its effect to the previous version
- The new "local" variables are not strictly necessary, but they make the code more easily *read* and more easily *changed*
- We should use our judgement to use local variables to help make our programs tidier, easier to read, more effective

Name clashes and scope

- Previously we saw a rule that said all variables must have different names
- This is not quite true:
  - Different methods *may* use the *same* names for local variables and parameters (this is a common practice)
- So the rule is:
  - "*Within any method* all the variable and formal parameter names must be different"
- It appears that there may be ambiguity if names are re-used. So we have the following "scope" rule:
  - Identifiers appearing in a method body *never* refer to the local variables or formal parameters of *another method*
- Further: When a method finishes ("returns to the caller"):
  - All its parameter and local variable memory is discarded
  - … and will be re-allocated at the next call

Variable naming and scope example

```
public class ScopeExample {
   public void method1(int x) {
      int y;
      ...x...y...          // Allowed – refers to
      int x;               // Not allowed
   }
   private void method2(int x, int y) {
      ...x...y...          // Allowed – refers to
      ...z...              // Not allowed
   }
   private void method3(int y) {
      int z;
      ...z...y...          // Allowed – refers to
      ...x...              // Not allowed
   }
}
```



Methods as building blocks

- If we design our new methods carefully, we can use our them as building block(s) for other new methods(s)
- Example: Perhaps we have designed a method to draw triangles

```
private void drawTriangle(Graphics drawingArea,
                          int xPlace, int yPlace,
                          int width, int height) {
  ... details omitted ...
}
```

Demo
TriangleMethod

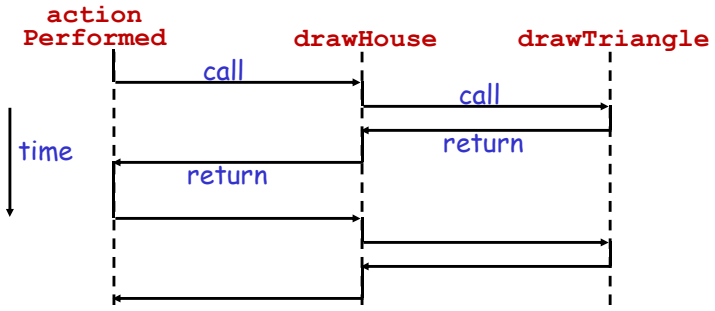
- Then we can design a method for drawing houses that calls our `drawTriangle` method to draw the roof

Demo
HouseDemo

```
private void drawHouse(Graphics drawingArea,
                       int topRoofX, int topRoofY,
                       int width, int height) {
    drawTriangle(drawingArea, topRoofX,
                 topRoofY, width, height);
    drawingArea.drawRect(topRoofX,
             topRoofY + height, width, height);

}
```


- We have traded-off a lengthy `actionPerformed` method against a more manageable design with a more "interesting" call/return sequence:



- In the `HouseDemo` program, we could technically do without the `drawTriangle` and `drawHouse` methods
  - But the `actionPerformed` method would be very long and confusing
  - And it would be awkward to add more houses!
  - With a collection of calls of `drawHouse`, with different actual parameter values, we can easily draw a variety of houses at different positions, with different sizes

Methods that return results (JFS, p73)

- Our methods so far are called for their "effect" or "action" only
  - Such methods have the word `void` in their headers
- Sometimes we would like a method to do a *computation*, and tell us the *result*
- In programming terms, this means
  - We give the method some values to work with (as parameters)
  - The method gives back ("returns") the computed result
- Two changes are required:
  - In the method header, the word `void` is replaced by a **type name** (the type of the value to be returned)
  - The method *body* must include a statement of the form:
    `return expression;`
    (the value which is returned is the current value of `expression`)

Returning results: Example method

- Example:

```
private int areaRectangle(int length,
                          int width) {
    int area;
    area = length * width;
    return area;
}
```

- Such a method must be called in a different way to our previous methods. This is because the method call

```
areaRectangle(15,27)
```

is *an expression,* it represents *a value of type* `int`, *not an instruction*

- Here is one way to call such a method:

```
fieldArea = areaRectangle(15,27);
```


Returning results: Example program

`AreaMethod` (JFS)

```
imports ... main ... createGUI ...


public void actionPerformed(ActionEvent event) {
    int a;
    a = areaRectangle(10, 20);
    JOptionPane.showMessageDialog(null,
                              "Area is:  " + a);
}
private int areaRectangle(int length,
                          int width) {
    int area;
    area = length * width;
    return area;
}
```

End of Section