Style in Programming (JFS Chap 19)

- In this section:
  - Why do we need to consider "style"
  - What makes a "quality" program?
  - Guidelines

Programming Style: Effective communication

- In a professional programming context it is likely that you will be working as a member of a team
- Various issues to consider:
  - You may have to develop code *jointly*
  - You may leave your job
  - You may fall under a bus
  - You may return to a program after several years on another project – you may have no detailed memory of it at all

A general principle

The text of a program should be presented so as to make it as easy as possible for a competent programmer to read and understand the program - *effective communication*

- Further: what helps the team can also help the individual!

Programming Style

- From the *user's* point of view, a quality program is one that:
  - Works correctly
  - Is easy to use, and reliable
- From the *programmer's* point of view, quality includes :
  - Understandability
  - Accessibility (e.g. navigability - finding things)
  - Maintainability (e.g. debugging and updating)
- It is normal to take into consideration
  - The layout of the text on the page/screen
  - The choice of names (identifiers)
  - Appropriate use of Java constructs
  - Use of methods to perform subtasks
  - The use of supporting text ("comments")

Some useful documents

- "Code Conventions for the Java Programming Language"

  `www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html`
- "How to Write Doc Comments for the Javadoc Tool"

  `www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Layout and Indentation

- Although a presentation style adopted is *essentially arbitrary*
  - The style chosen should be applied *consistently*
- We have been using the Java for Students "house style" – *a typical professional style.*
- Key features:
  - *Sequences* of related items have their *first characters aligned vertically*
  - `{` at *end of the first line* of a structure;
    `}` on a line by itself, *directly below* the first character of the structure
  - Items within a class, a method, `if` statement *branches,* and *loop bodies* are indented consistently by "n" *further* spaces (say 2, 3 or 4 spaces)
  - Blank lines are used to separate identifiable sections of code

Choosing Identifiers

- Good choice of identifiers is a straightforward way to make a program more readable:
  - Choose informative names for variables, methods, classes
  - They should have *mnemonic* value
- Classes should usually be *nouns* – they represent *entities*
  - Eg: `StudentRecord`, `Button`, `Document`
- Void methods should usually be *verbs* – they represent *actions*
  - Eg: `paintScreen`, `setValue`, `addActionListener`, `setEnabled`
- Non-void methods are usually *queries*
  - Eg: `getValue`, `isEnabled`, `getName`, `hasGraduated`
- Variables are usually *nouns* – they represent *entities* or *information*
  - Eg: `fontSize`, `carCount`, `growButton`, `word`

- For local, small-scale or temporary use:
  - It is OK to use sensible abbreviations such as **`max`, `tot`**
  - and conventional names such as **`i`**, **`j`**, **`x`**, **`y`**
- Adopt a consistent style as regards the use of upper/lower case and underscores. Our style is
  - No underscores
  - Variable and method names start with lower case
  - Class names start with upper case
  - Subsequent word segments start with upper case, and all other letters are lower case
  - Exception: Special constants have all upper case names Eg: **`JSlider.HORIZONTAL`**, **`Color.BLACK`**
  - *This makes it easier to see what kind of item the identifier refers to*

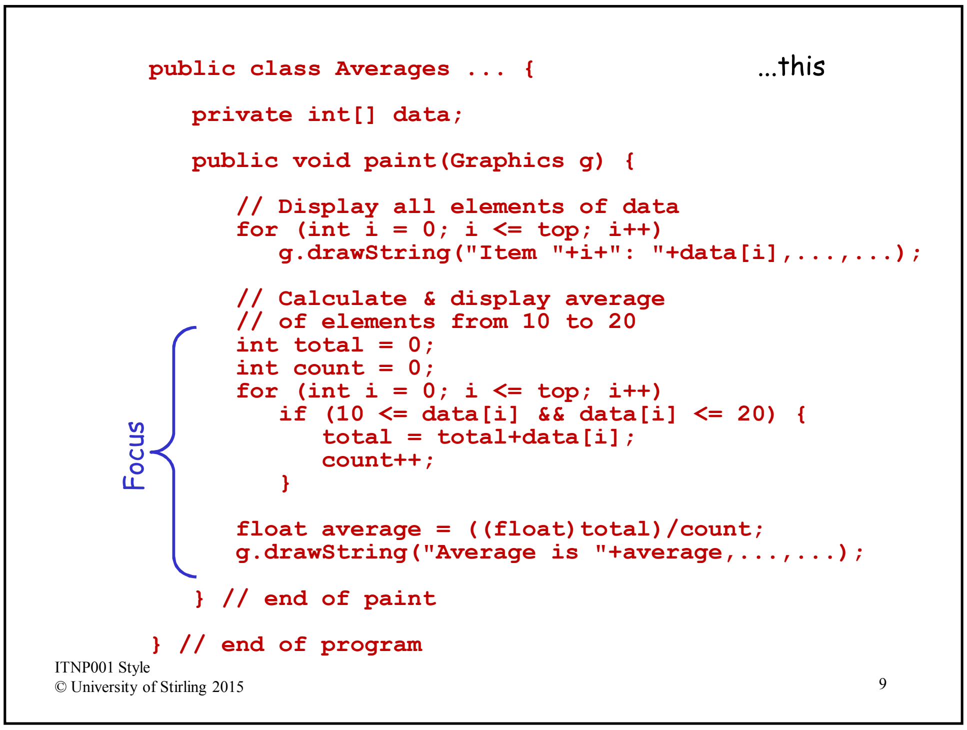
Alignment, Indentation Patterns, Spacing, Identifiers

Compare and contrast this with...

```
public class Myprog ... {
private int[] m;
public void paint(Graphics p) {
// Display all elements of m
for (int z = 0; z <= top; z++)
p.drawString("Item "+z+": "+m[z],...,...);
// Calculate & display average
// of elements from 10 to 20
int qr = 0;
int g = 0;
for (int zz = 0; zz <= top; zz++)
if (10 <= m[zz] && m[zz] <= 20) {
qr = qr+m[zz];
g++;
}
float tt = ((float)qr)/g;
p.drawString("Average is "+tt,...,...);
} // end of paint
} // end of program
```

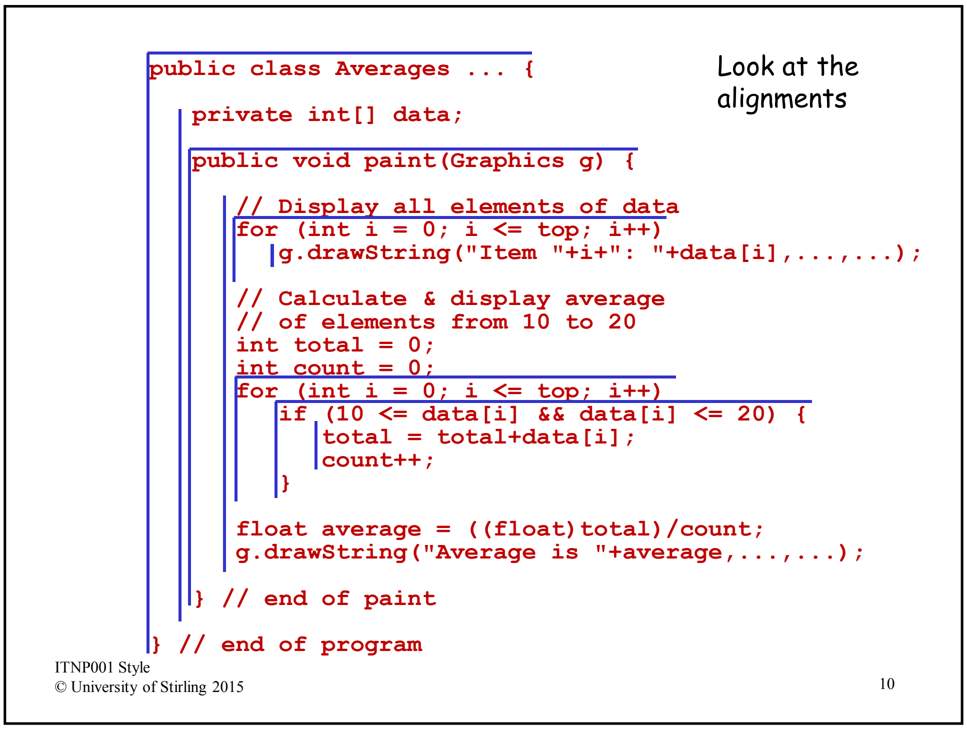
Focus

!



…this

```
public class Averages ... {

   private int[] data;

   public void paint(Graphics g) {

      // Display all elements of data
      for (int i = 0; i <= top; i++)
         g.drawString("Item "+i+": "+data[i],...,...);

      // Calculate & display average
      // of elements from 10 to 20
      int total = 0;
      int count = 0;
      for (int i = 0; i <= top; i++)
         if (10 <= data[i] && data[i] <= 20) {
            total = total+data[i];
            count++;
         }

      float average = ((float)total)/count;
      g.drawString("Average is "+average,...,...);

   } // end of paint

} // end of program
```



Look at the alignments

```
public class Averages ... {

   private int[] data;

   public void paint(Graphics g) {

      // Display all elements of data
      for (int i = 0; i <= top; i++)
         g.drawString("Item "+i+": "+data[i],...,...);

      // Calculate & display average
      // of elements from 10 to 20
      int total = 0;
      int count = 0;
      for (int i = 0; i <= top; i++)
         if (10 <= data[i] && data[i] <= 20) {
            total = total+data[i];
            count++;
         }

      float average = ((float)total)/count;
      g.drawString("Average is "+average,...,...);

   } // end of paint

} // end of program
```

Methods

- A well-designed program uses methods for individual sub-tasks
  - This aids initial construction, readability, and later maintainability
- Methods can be used to avoid duplicated code
  - But they are valuable even if called in only one place!
- Separate subtasks should not be combined in one method
  - Some judgement is required to identify appropriate subtasks
- Methods should not be too long
  - 40 lines is becoming too long, and ideally methods should be less than half that
- Methods should have appropriate names, and comments should explain their purpose
- Well chosen method names can reduce the need for comments in the *calling* code

Comments

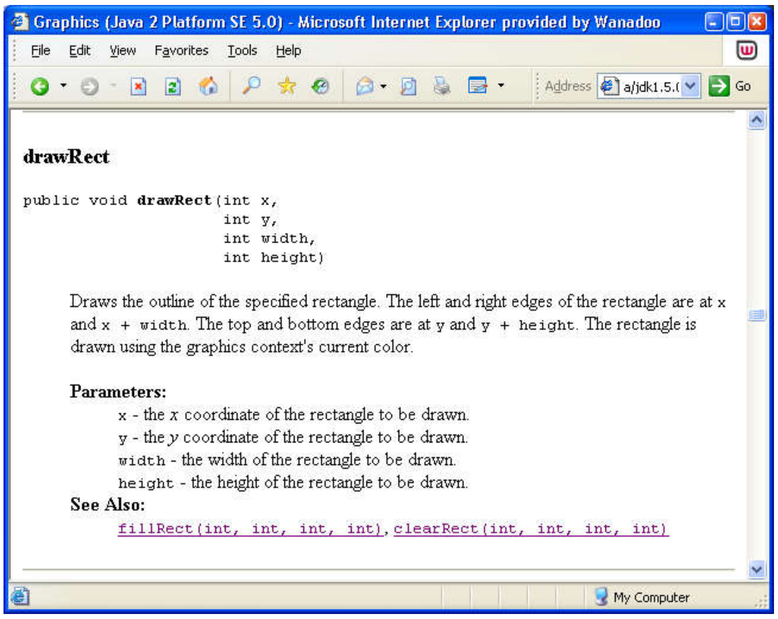
- Comments are informal text added to the source text of programs
- Effective commenting requires practice
  - Requires a good feel for the kind of detail that the **reader** needs
- Comments should give *overviews* and *information not readily available in the code itself*
- The intelligent reader needs *signposts* and *hints*
  - Not impenetrable detail,
  - Nor a wilderness
- Don't include comments which make *obvious* points
  - Do not re-state what can be easily seen from the code
  - Eg: Poor: `count = count+1; // Add one to count`
    Better: `count = count+1; // One more car`

Where should comments go?

- Classes should have an introductory (header) comment:
  - Stating the purpose of the program.
  - Summarizing important technical detail.
  - Including the date and author's name
  - Possibly including the update history
- Variable declarations should have a brief comment describing the role of each variable
- Methods should have introductory comments, to explain their purpose and use
  - Comments may be used to describe the roles of formal parameters
- "Paragraphs" within methods may have introductory comments
- Individual statements may have optional comments to the right

"Doc comments", the javadoc tool, and automatically generated documentation

- Sun has devised special stylised comments
  - "Documentation comments" or "doc comments"
  - (Sometimes incorrectly called "javadoc comments")
- And supplies a program, the javadoc tool, included with the JDK
- javadoc processes Java files containing doc comments and generates nicely formatted HTML pages describing the program
- For example: All the Java libraries are documented in this way
  - You may have seen the **`drawRect`** description in the on-line **`Graphics`** library page (see next slide)
  - This was produced by javadoc from the library's source code (see slide after next)


Graphics (Java 2 Platform SE 5.0) - Microsoft Internet Explorer provided by Wanadoo
File Edit View Favorites Tools Help
Address a/jdk1.5.(Go
My Computer

drawRect

```
public void drawRect(int x,
                     int y,
                     int width,
                     int height)
```

Draws the outline of the specified rectangle. The left and right edges of the rectangle are at `x` and `x + width`. The top and bottom edges are at `y` and `y + height`. The rectangle is drawn using the graphics context's current color.

Parameters:
`x` - the *x* coordinate of the rectangle to be drawn.
`y` - the *y* coordinate of the rectangle to be drawn.
`width` - the width of the rectangle to be drawn.
`height` - the height of the rectangle to be drawn.

See Also:
`fillRect(int, int, int, int)`, `clearRect(int, int, int, int)`

- The method starts with this comment:

```
/**
  * Draws the outline of the specified rectangle.
  * The left and right edges of the rectangle are at
  * <code>x</code> and <code>x&nbsp;+ ...
  * The top and bottom edges are at
  * <code>y</code> and <code>y&nbsp;+ ...
  * The rectangle is drawn using the graphics ...
  *
  * @param         x   the <i>x</i> coordinate ...
  * @param         y   the <i>y</i> coordinate ...
  * @param         width   the width of the rectangle
  * @param         height   the height of the
  *
  * @see          java.awt.Graphics#fillRect
  * @see          java.awt.Graphics#clearRect
  */
public void drawRect(int x, int y,
                        int width, int height) {
```

The basic form of doc comments

- Doc comments *immediately precede the item they describe*
- They may contain embedded HTML
- Classes start with:

```
/**
 * A description of class Xyzw here
 *
 * @author  (your name)
 * @version (a version number or a date)
 */
public class XYZW ...
```

  - BlueJ *Edit menu/New Class* creates a template
- Global variables are described like this:

```
/**
 * Description of role of variable abc
 */
private int abc;
```

- Methods start with:

```
/**
 * Comment describing sampleMethod
 *
 * @param  y  a sample parameter for a method
 * @return    the square of y
 */
public int sampleMethod(int y) {
```

  - BlueJ *Edit menu/Insert method* creates a template
- BlueJ can run javadoc for you (via Tools menu/Project Documentation) – and it launches a browser
  - Unfortunately it does not process `private` items by default
  - (Coercing it to do so is possible but awkward)

End of section