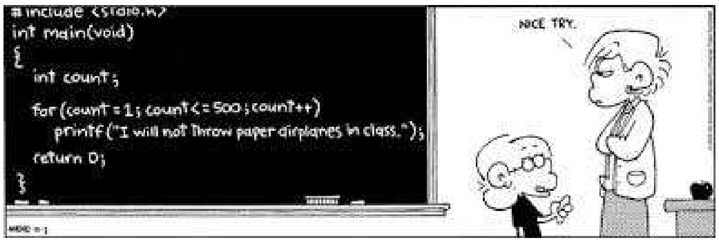


```
#include <stdio.h>
int main(void)
{
  int count;
  for (count = 1; count <= 500; count++)
    printf("I will not Throw paper airplanes in class.");
  return 0;
}
```

Clean Formatting

www.cs.uoi.gr/~zarras/soft-devII.htm

from Clean Code by R. C. Martin, a.k.a "Uncle Bob"

The Purpose of Formatting

The purpose of a computer program is to tell other people what you want the computer to do. – Donald Knuth

The purpose of formatting is to facilitate communication. The formatting of code conveys information to the reader.

Vertical & Horizontal Formatting

In code format we distinguish 2 aspects:

Vertical formatting which concerns blocks of lines of code and refers to the way code is structured from top to bottom

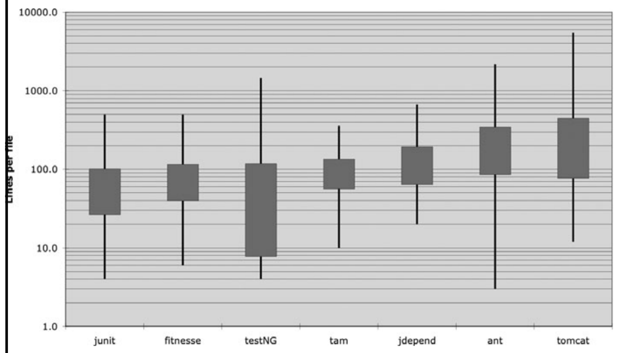
Horizontal formatting which concerns a single line of code and refers to the way it is structured from left to right

Vertical Formatting

How big should a source file be?
How big are most Java source files?

Vertical Formatting

How big should a source file be?
How big are most Java source files?



JUnit, FitNesse, and T&M are composed of relatively small files. None are over 500 lines and most of those files are less than 200 lines.

Tomcat and Ant, on the other hand, have some files that are several thousand lines long and close to half are over 200 lines.

So what is the conclusion ?

Vertical Formatting

It appears to be **possible to build <u>significant systems</u>** (FitNesse is close to 50,000 lines) out of files that are **typically 200 lines long**, with an upper limit of 500.

Although this should not be a hard and fast rule, it should be considered very desirable.

Small files are usually easier to understand than large files are !!!

How to structure the file vertically ? The Newspaper Metaphor

We would like a source file to be like a newspaper article.

Think of a well-written newspaper article.

You read it vertically.

At the top you expect a headline that will tell you what the story is about and allows you to **decide** whether it is something you want to read.

The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts.

As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other.

THE GOOD NEWS
7 BILLION PEOPLE CONTINUE TO LIVE
50,000 FLIGHTS SAFELY TAKE OFF AND LAND
NEW DRUGS AND CURES FOR ILLNESS BEING INVENTED EVERY DAY
KITTENS
WWW.LIZZLIZZ.COM

How to structure the file vertically ? The Newspaper Metaphor

We would like a source file to be like a newspaper article.

The **file name** should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not.

The **topmost parts** of the source file should provide the high-level concepts and algorithms.

Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.

THE GOOD NEWS
7 BILLION PEOPLE CONTINUE TO LIVE
50,000 FLIGHTS SAFELY TAKE OFF AND LAND
NEW DRUGS AND CURES FOR ILLNESS BEING INVENTED EVERY DAY
KITTENS
WWW.LIZZLIZZ.COM

The Newspaper Metaphor

We would like a source file to be like a newspaper article.

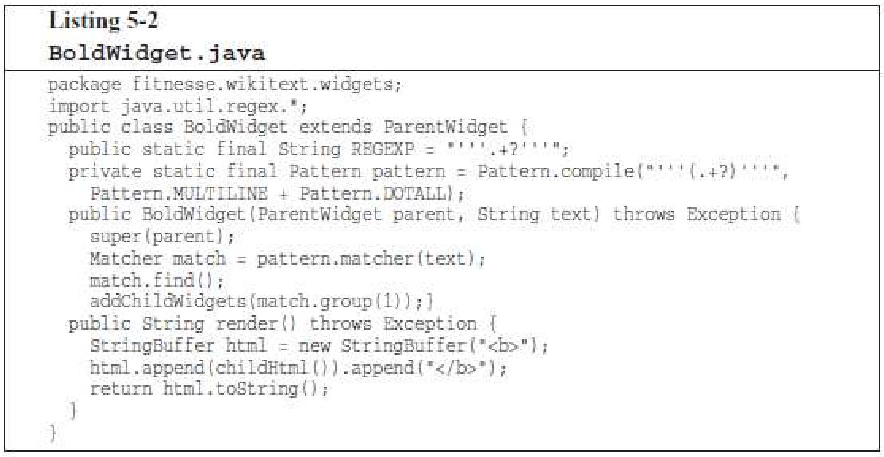
A newspaper is composed of many articles; most are very small.

Some are a bit larger.

Very few contain as much text as a page can hold.

THE GOOD NEWS
7 BILLION PEOPLE CONTINUE TO LIVE
50,000 FLIGHTS SAFELY TAKE OFF AND LAND
NEW DRUGS AND CURES FOR ILLNESS BEING INVENTED EVERY DAY
KITTENS
WWW.LIZZLIZZ.COM

Vertical Openness



Listing 5-2

BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Take a look for a while !!

Vertical Openness

What is the name of the class ?

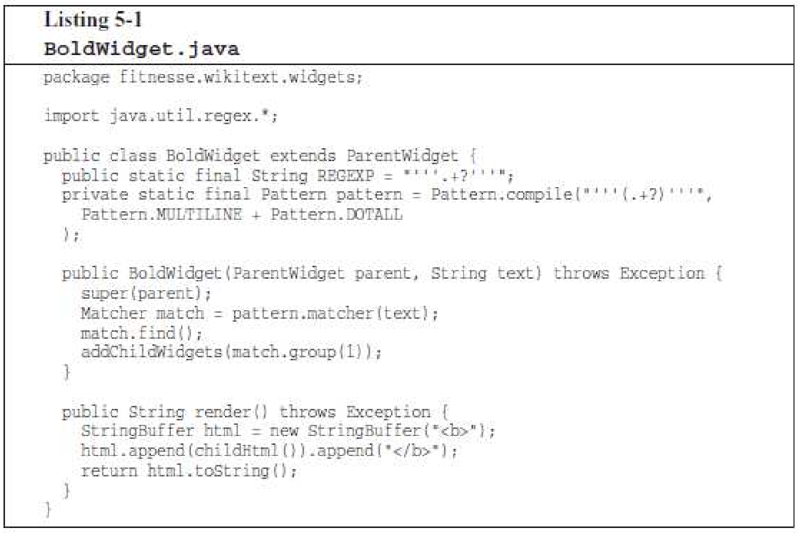
What is the package it belongs too ?

How many imports does it have ?

How many methods ?

How many attributes ?

Vertical Openness



Listing 5-1

BoldWidget.java

```java
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Same code much better formatted !! The difference is vertical openness

Vertical Openness

Vertical openness means separate the concepts with blank lines

A piece of code describes a number of concepts:

For a class we have:

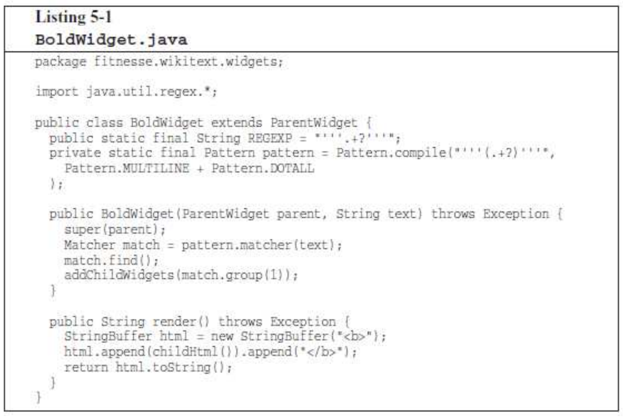
the **package** where it belongs

other classes that are **used/imported** (i.e. **dependencies**)

the **state** of the class objects (i.e. the set of fields)

each one of the **methods** that constitute the behavior

Vertical Openness


Listing 5-1
BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

There are **blank lines** that separate the package declaration, the import(s), and each of the functions.

This extremely **simple rule** has a profound **effect** on the **visual layout** of the code.

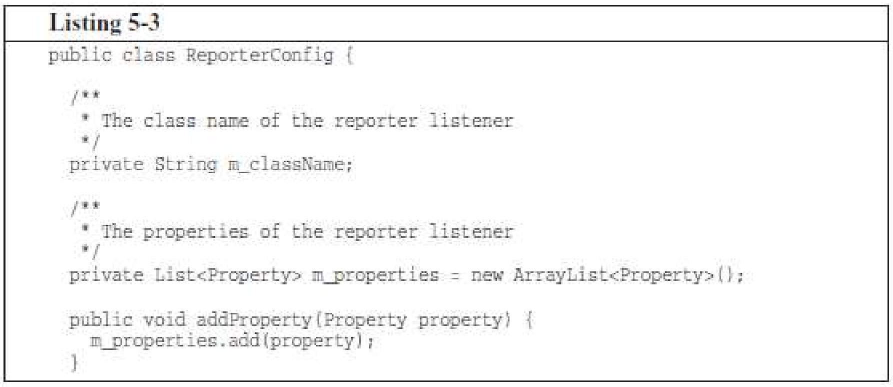
Each **blank line** is a visual cue that **identifies** a new and separate **concept**.

As you scan down the listing, your eye is drawn to the first line that follows a blank line.

Take a look for a while !!

Same code much better formatted !!
The difference is a bit of vertical openness

Vertical Density

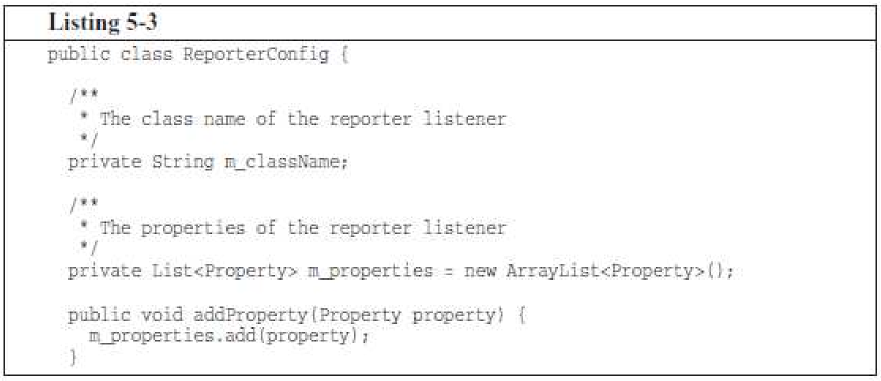


Listing 5-3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Take a look for a while !! Do you see the inconvenience here ??

Vertical Density



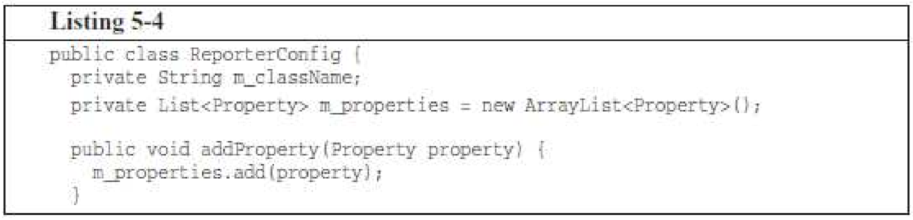
Listing 5-3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Take a look for a while !! Do you see the inconvenience here ??

Notice how the useless comments break the close association of the two attributes that constitute the state of the class objects !!!

Vertical Density



Listing 5-4

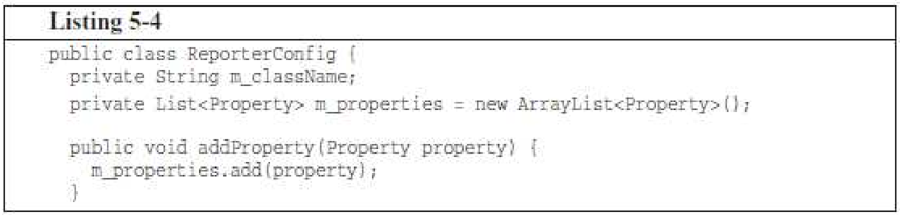
```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Much easier to read.

It fits in an "eye-full". We can look at it and see that this is a class with two attributes and a method, without having to move our head or eyes much.

The previous listing forces us to use much more eye and head motion to achieve the same level of comprehension.

Vertical Density



Listing 5-4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

What have we done ??

We increased the vertical density, by putting together lines of code that belong to the **same concept** (i.e. the **state definition** of the class objects)

So lines of code that are tightly related (defining a concept) should appear vertically dense.

Vertical Distance

Vertical distance is the distance between concepts that relate with each other, like methods that call each other

Concepts that are closely related should be kept in a small vertical distance!!

Vertical Distance

```
private static void readPreferences() {
   InputStream is= null;
   try {
      is= new FileInputStream(getPreferencesFile());
      setPreferences(new Properties(getPreferences()));
      getPreferences().load(is);
   } catch (IOException e) {
      try {
         if (is != null)
            is.close();
      } catch (IOException e1) {
      }
   }
}
```

Variables should be declared as **close** to their **usage** as possible.

Because our functions are very **short**, local variables could appear at the **top** of each function, as in this function from JUnit

Vertical Distance

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
     count += each.countTestCases();
  return count;
}
```

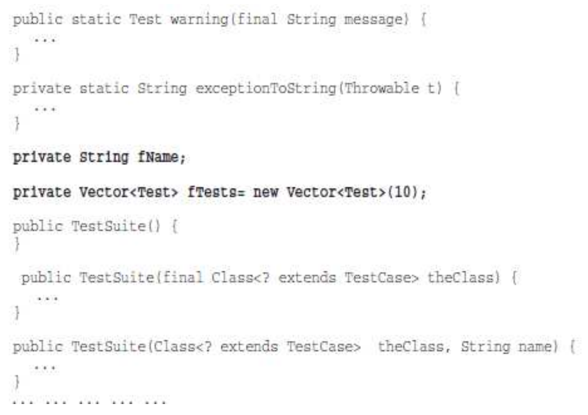
Control variables for loops should usually be declared **within** the **loop** statement, as in this function from JUnit.

Vertical Distance

```
public static Test warning(final String message) {
   ...
}

private static String exceptionToString(Throwable t) {
   ...
}

private String fName;

private Vector<Test> fTests= new Vector<Test>(10);

public TestSuite() {
}

 public TestSuite(final Class<? extends TestCase> theClass) {
   ...
}

public TestSuite(Class<? extends TestCase>  theClass, String name) {
   ...
}
... ... ... ... ...
```

How about these for class attribute declarations (from the Junit TestSuite class) ??

Vertical Distance



```
public static Test warning(final String message) {
   ...
}
private static String exceptionToString(Throwable t) {
   ...
}
private String fName;

private Vector<Test> fTests= new Vector<Test>(10);
public TestSuite() {
}
 public TestSuite(final Class<? extends TestCase> theClass) {
   ...
}
public TestSuite(Class<? extends TestCase>  theClass, String name) {
   ...
}
... ... ... ... ...
```

How about this for class attribute declarations (from the Junit TestSuite class) ??

Two attributes declared in the middle of the class !! It would be hard to hide them in a better place. Someone reading this code would have to stumble across the declarations by accident **!!!**

Vertical Distance

But putting the attributes at the top (or bottom) shall increase **the vertical distance** between them and the functions that use them ???

Vertical Distance

But putting the attributes at the top (or bottom) shall increase **the vertical distance** between them and the functions that use them ???

Not really, because in a well-designed – cohesive - class, they are used by many, if not all, of the methods of the class !!!

Vertical Distance

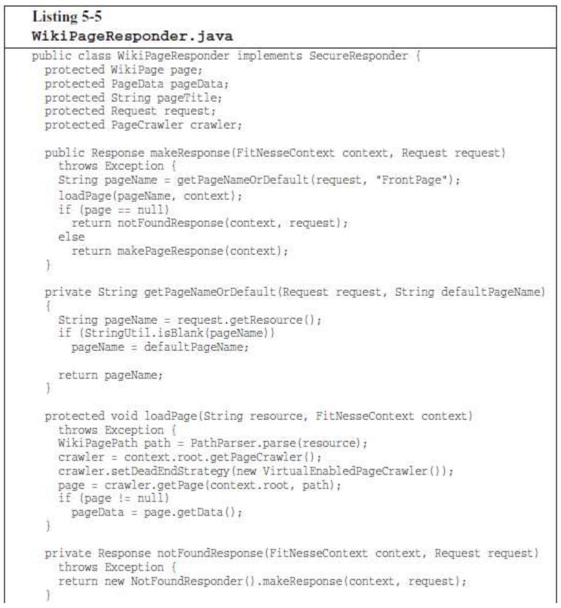
There have been many debates over where class attributes should go.

In C++ we commonly practiced the so-called *scissors rule, which put all the attributes at the* bottom.

In Java, however, is to put them all at the top of the class.

The important thing is for the attributes to be declared in one well-known place. Everybody should know where to go to see the declarations !!!

Vertical Distance

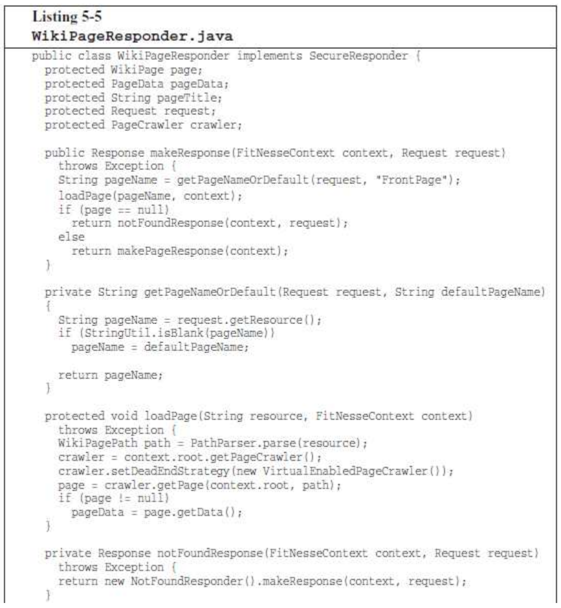


Listing 5-5
WikiPageResponder.java

```java
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }
```

What do we observe here ??

Vertical Distance



Listing 5-5
WikiPageResponder.java

```java
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }
```

What do we observe here ??

If one **function calls another**, they should be vertically **close**,

and the **caller** should be **above** the **callee**, if at all possible.

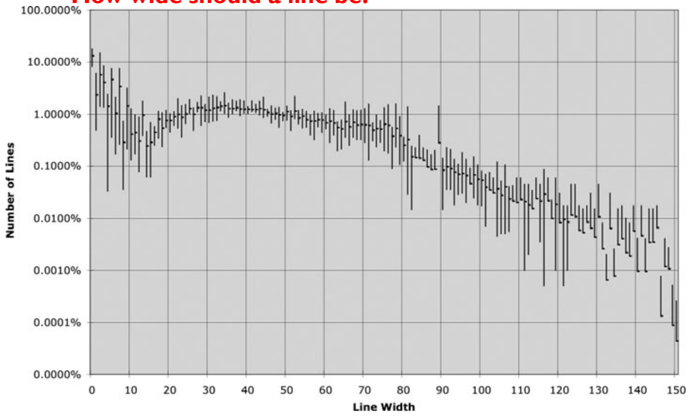
This gives the program a natural flow, according to the newspaper metaphor., (a.k.a **step-down rule**)

Horizontal Formatting

How wide should a line be?

Horizontal Formatting

How wide should a line be?



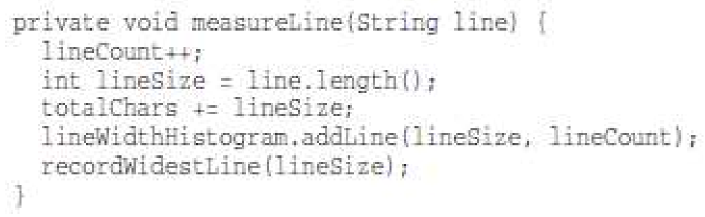
Every size from **20 to 60** represents about 1% of the total number of lines.

That's **40 percent**!

Another 30 percent are **less than 10** characters wide.

Programmers clearly prefer short lines !!!

Horizontal Openness & Density



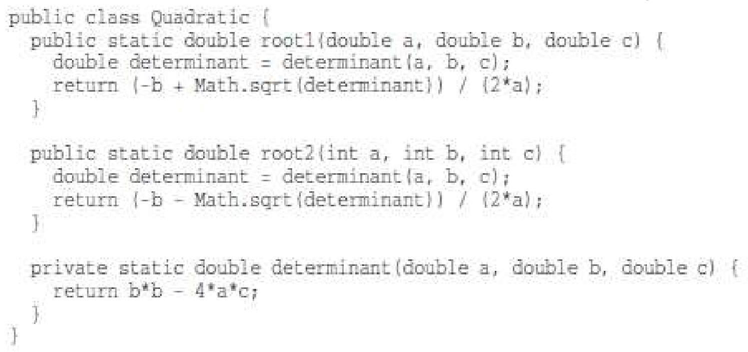
```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Surround the assignment operators with white space to accentuate them. Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

On the other hand, don't put spaces between the function names and the opening parenthesis. This is because the function and its arguments are closely related.

Separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.

Horizontal Openness & Density



```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Notice how nicely the equations read.

The factors have no white space between them because they are high precedence.

The terms are separated by white space because addition and subtraction are lower precedence.

Indentation

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

How about this as horizontal formatting ??

Indentation

```
public class FitNesseServer implements SocketServer {
  private FitNesseContext context;

  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

A source file is a **hierarchy** rather like an **outline**. There is information that pertains to the file as a whole, to the individual classes within the file, to the methods within the classes, to the blocks within the methods, and recursively to the blocks within the blocks.

To make this hierarchy of scopes visible, **we indent** the lines of source code in proportion to their position in the hierarchy.

Horizontal Alignment

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket         socket;
  private   InputStream    input;
  private   OutputStream   output;
  private   Request        request;
  private   Response       response;
  private   FitNesseContext context;
  protected long           requestParsingTimeLimit;
  private   long           requestProgress;
  private   long           requestParsingDeadline;
  private   boolean        hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Once upon a time….!!!
Do you see the problem here ??

Horizontal Alignment

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket         socket;
  private   InputStream    input;
  private   OutputStream   output;
  private   Request        request;
  private   Response       response;
  private   FitNesseContext context;
  protected long           requestParsingTimeLimit;
  private   long           requestProgress;
  private   long           requestParsingDeadline;
  private   boolean        hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

The alignment seems to emphasize the wrong things and leads the eye away from the true intent.

In the list of declarations above we are tempted to read down the list of variable names without looking at their types.

Likewise, in the list of assignment statements we are tempted to look down the list of rvalues without ever seeing the assignment operator.

Indentation

```
while((result = dis.read()) != null);
System.out.println(result);
```

What does this do ??

Indentation

```
while((result = dis.read()) != null);
System.out.println(result);
```

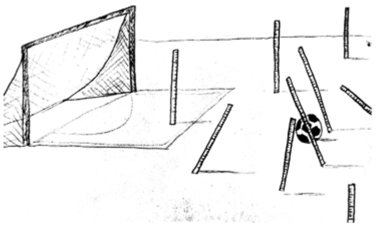
What does this do ??

Sometimes the body of a while or for statement is a dummy !!

Unless you make that semicolon *visible by* ***indenting it*** *on it's own line, it's* just too hard to see.

Team Rules

Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules.



A team of developers should agree upon a **single formatting style**, and then every member of that team should use that style.

We want the software to have a consistent style.

We don't want it to appear to have been written by a bunch of disagreeing individuals.

Java Coding Standard Style

Oracle

Java Conventions

- **Source file organization**

Java Conventions

- **Source file organization**
  - A Java source file should contain the following elements, in the following order:
    - Copyright/ID block comment
    - package declaration
    - import declarations
    - one or more class/interface declarations
  - At least one blank line should separate all of these elements.

Java Conventions

- **Package naming**

Java Conventions

- **Package naming**
  - Generally, package names should use only **lower-case letters** and digits, and no underscore. Examples:
    - `java.lang`
    - `java.awt.image`
    - `dinosaur.theropod.velociraptor`

Java Conventions

- **Class/Interface naming**

Java Conventions

- **Class/Interface naming**
- All type names (classes and interfaces) should use the ***InfixCaps*** style.
  - Class names should be **nouns** or **noun phrases**.
  - Start with an **upper-case** letter, and **capitalize the first letter of any subsequent word** in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case.
  - Do not use underscores to separate words.
  - Examples:
    - `// GOOD type names:`
      - `LayoutManager, ArrayIndexOutOfBoundsException`
    - `// BAD type names:`
      - `ManageLayout // verb phrase`
      - `awtException // first letter lower-case`
      - `array_index_out_of_bounds_exception // underscores`

Java Conventions

- **Field naming**

Java Conventions

- **Field naming**
- The names should be **nouns** or **noun phrases**.
- Names of **non-constant fields** (reference types, or non-final primitive types) should use the *infixCaps* style.
  - Start with a **lower-case** letter, and **capitalize the first letter of any subsequent word** in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case.
  - Do not use underscores to separate words.
  - Examples:
    - `char recordDelimiter;`
- Names of fields being used as ***constants*** should be **all upper-case**, with **underscores** separating words.
  - Examples:
    - `MIN_VALUE, MAX_BUFFER_SIZE, OPTIONS_FILE_NAME`

Java Conventions

- **Method naming**

Java Conventions

- **Method naming**
- Method names should be **imperative verbs** or **verb phrases**. Examples:
- Method names should use the *infixCaps* style.
  - Start with a **lower-case** letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case.
  - Do not use underscores to separate words.
  - // GOOD method names:
    - `showStatus(), drawCircle(), addLayoutComponent()`
  - // BAD method names:
    - mouseButton() // noun phrase; doesn't describe function
    - DrawCircle() // starts with upper-case letter
    - add_layout_component() // underscores

Java Conventions

- **Blank lines**

Java Conventions

- **Blank lines**
  - A blank line should also be used in the following places:
    - After the copyright block comment, package declaration, and import section.
    - Between class declarations.
    - Between method declarations.
    - Between the last field declaration and the first method declaration in a class.

Java Conventions

- **Blank spaces**
- **A single blank space (not tab) should be used:**
  - Between a **keyword** and its opening **parenthesis**. This applies to the following keywords: `catch, for, if, switch, while.`
  - After any **keyword** that takes an **argument**.
    - Example: `return true;`
  - Between **two adjacent keywords**.
  - Before *and* after **binary operators except .(dot).**
  - After a **comma** in a list.
  - After the **semi-colons** in a for statement, e.g.:
    - `for (expr1; expr2; expr3) {`

Java Conventions

- **Continuation lines**
  - Lines should be limited to 80 columns.
  - Lines longer than 80 columns should be **broken into one or more continuation lines**, as needed.
  - All the continuation lines should be aligned
    - // RIGHT

      -
        ```
        foo(long_expression1, long_expression2, long_expression3,
            long_expression4);
        ```
    - // RIGHT

      -
        ```
        foo(long_expression1,
            long_expression2,
            long_expression3,
            long_expression4);
        ```