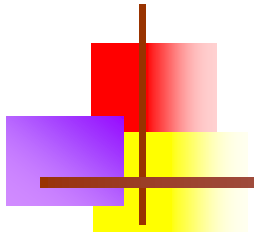
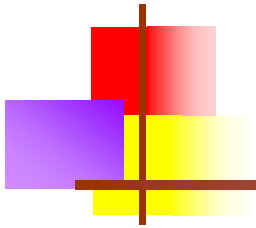
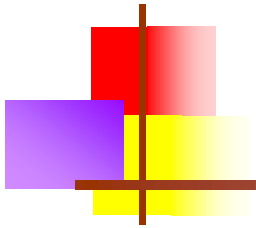


# Error messages



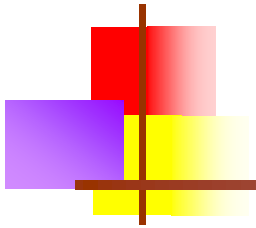
# Types of errors

- There are three general types of errors:
  - Syntax (or "compile time") errors
    - Syntax errors are "grammatical" errors and are detected when you compile the program
    - Syntax errors prevent your program from executing
  - Runtime errors
    - Runtime errors occur when you tell the computer to do something illegal
    - Runtime errors may halt execution of your program
  - Logic errors
    - Logic errors are not detected by the computer
    - Logic errors cause your results to be wrong



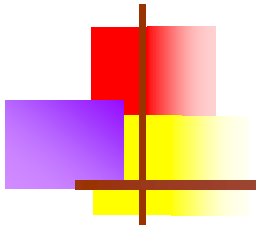
# Syntax errors

- A syntax error is a "grammatical" error--bad punctuation, misuse of keywords, etc.
- Syntax error messages tell you two things:
  - The line number at which an error was detected
    - *Usually*, this is the line containing the error
    - In some cases, the actual error is *earlier* in the program text
    - Use an editor that shows line numbers!
  - What the compiler *thinks* the problem is
    - Since the compiler cannot know what you meant, the message is only a "best guess," and is sometimes misleading
- Syntax errors can cascade: An early error can cause spurious error messages later in the program
  - Always fix the earliest message first
  - If later messages don't make sense, try recompiling



# Example syntax errors

- System.out.println("(g1 == g2) = " + (g1 == g2);
  - ')' expected
- System.out.println("(g1 == g2) = " + (g1 == g2)));
  - ';' expected
- a = g1 + g2;
  - cannot resolve symbol -- variable a
    - a was never declared; *or*
    - a was declared, but in some other scope; it's not visible here
- a = 5;
  - <identifier> expected
    - This is a statement, and statements can only occur inside methods
- a = b;
  - variable b might not have been initialized



# Runtime errors

- A runtime error occurs when your program does something illegal
  - Runtime errors typically stop your program
  - Runtime errors can be “caught” by your program, thus allowing the program to continue running
    - We’ll discuss how to do this later in the course
  - Runtime errors are usually caused by something the program did (or failed to do) earlier in execution
    - Because the *cause* of the error is somewhere else in the program, runtime errors are usually harder to solve

# A common runtime error

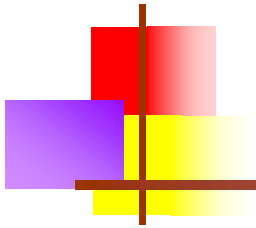
```
java.lang.NullPointerException
    at Test.run(Test.java:22)
    at Test.main(Test.java:6)
    at __SHELL1.run(__SHELL1.java:6)
    at bluej.runtime.ExecServer.suspendExecution(ExecServer.java:187)
    at bluej.runtime.ExecServer.main(ExecServer.java:69)
```

} What kind of error it was

Traceback: How your program got to where the error was detected (in line 22 of the file Test.java)

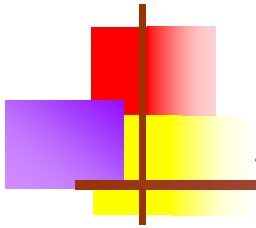
The part of the traceback in your code (line 22 of the file, in your run method, called from line 6 of the file, in your main method)

The part of the traceback in the Java and Eclipse system; you can pretty much ignore this part



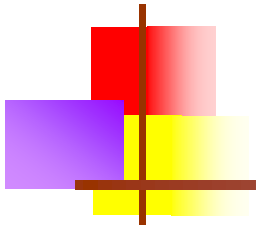
# Null pointer exceptions

- The NullPointerException is one of the most common runtime errors
  - It occurs when you send a message to a null variable (a non-primitive variable that doesn't refer to an actual object)
  - The null variable causing the problem is *always* just before a dot
  - Example: g.drawLine(x1, y1, x2, y2);
    - If this caused a NullPointerException, the variable g *must* have been null
    - You probably never initialized g
    - Java *tries* to catch uninitialized variables, but it cannot catch them all



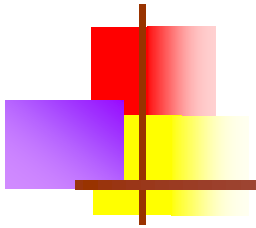
# Assertion errors

- assert 2 + 2 == 5;
  - Exception in thread "main" java.lang.AssertionError
    at Test.run(Test.java:9)
    at Test.main(Test.java:6)
- assert 2 + 2 = 5: "2 + 2 is actually " + (2 + 2);
  - Exception in thread "main" java.lang.AssertionError:
    2 + 2 is actually 4
    at Test.run(Test.java:9)
    at Test.main(Test.java:6)
- Use assert statements to tell what you believe will always be true at a given point in the program
  - assert statements are best used as an "early warning system," rather than as a debugging tool once errors are known to happen
  - assert statements are also valuable as documentation



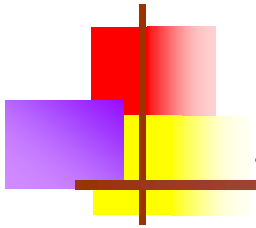
# Logic errors

- A logic error is when your program compiles and runs just fine, but does the wrong thing
- In *very simple* programs, logic errors are usually easy to find and fix, if they occur at all
- In all but the simplest of programs,
  - 10% of your time is spent writing the program and fixing the syntax errors (more if you are still learning the syntax)
  - 90% of your time is spent finding and fixing runtime and logic errors
- Logic errors can take hours, or even days, to find
  - Allocate your time accordingly!



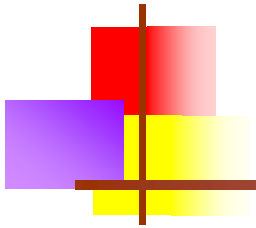
# Make better use of your time

- While you cannot *avoid* making errors, you can *prepare* for them
  - Keep your programs as *simple* as possible
  - *Indent* properly so you can see the structure of your code
  - *Comment* your programs so you can find things again
  - Write *test cases* and use them
    - "Write a little, test a little"
  - Write assert statements for the things that you "know" cannot possibly happen
- Programming is a creative activity, and can be very enjoyable and satisfying
  - For most of us, debugging is *not* the fun part



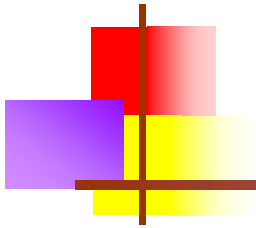
# Approaches to finding errors

- Try a random change and see if it helps
  - "An infinite number of monkeys, given an infinite number of typewriters, will eventually produce the complete works of Shakespeare"
  - No monkey has ever passed this course
  - You can't afford this approach--it's stupid and doesn't work
- Better approach: *Think* about what could have caused the results you see, and then *look* for where that might have occurred
  - Advantage: Leads you directly to the error
  - Disadvantage: Not always possible to figure out what could have happened



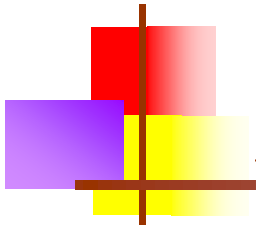
# Good approaches, I

- Put in print statements
  - Advantage: Helps you see what the program is doing
  - Disadvantage: You have to take them out again
- Use a Debugger
  - A debugger is a program that lets you step through your program line by line and see exactly what it is doing
  - Advantages:
    - Takes no prior preparation
    - Lets you see exactly what the program is doing
  - Disadvantage
    - You have to learn to use one (but it's worth the trouble!)



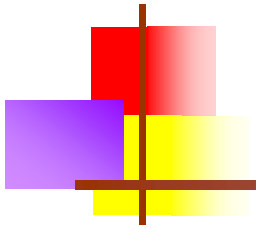
# Good approaches, II

- Explain your code to a friend (even if your friend can't program)
  - Advantage: Forces you to actually *look* at what you did
  - Disadvantage: Requires you to have friends
    - In a pinch, even your dog will do (if you can get him to hold still long enough)
  - Note: This is *not* a violation of academic honesty!
- In extremis: *Throw out* the offending code and rewrite it
  - Advantages:
    - Usually works
    - Usually makes your code simpler
  - Disadvantage: Can be psychologically difficult



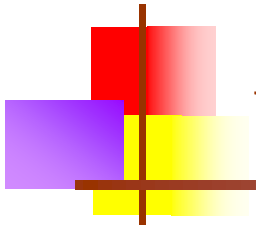
# A “best” approach

- According to the Extreme Programming (XP) philosophy, you should write a suite of tests *before* you write the code
  - This helps clarify what the code is supposed to do
    - It’s a good idea to know this before starting to program!
  - Having a test suite also makes it much less risky to change and improve working code
    - The mantra is: “Write a little, test a little.”
    - That is, make changes in small steps, and ensure after each step that the code still works
    - This only makes sense if running the tests is fast and simple
  - Use JUnit to build and run your test suites



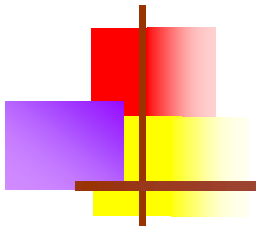
# Testing is *your* responsibility

- You are expected to turn in correct programs, or as nearly correct as you can make them
- Testing is a vital part of writing programs
- Test your program, not only with "typical" data, but also all the weird and unusual cases that you can think of
- When I supply example data, it is *only* to help explain what your program should do; you should *never* think of such data as an adequate test set
- We will usually test your program with data you have never seen
- To get 100%, your programs must pass every test *and* have good style
- Programs with syntax errors are worth *zero* points
  - (You've done less than 10% of the work)



# Why such high standards?

- Large programs are the most complex entities ever devised by mankind
  - Large programs consist of thousands of methods and hundreds of thousands of lines
  - A single error can cause catastrophic failure
  - There are a great many examples, from space vehicles crashing to deaths from badly programmed medical equipment
  - We give partial credit for “mostly working” programs--and, believe it or not, I think this is *generous*



# The End (for now)

"To err is human, but to really foul things up requires a computer."

Farmer's Almanac, *1978*