Objective

Testing and Debugging

- To learn how to carry out unit tests
- To understand the principles of test case selection and evaluation
- To become familiar with using a debugger
- To learn strategies for effective debugging

Study sections 2.7, 3.4, 5.6, 6.10, 7.8, 8.7, and 10.6 of BigJava: Early Object 6th edition.

Debugging

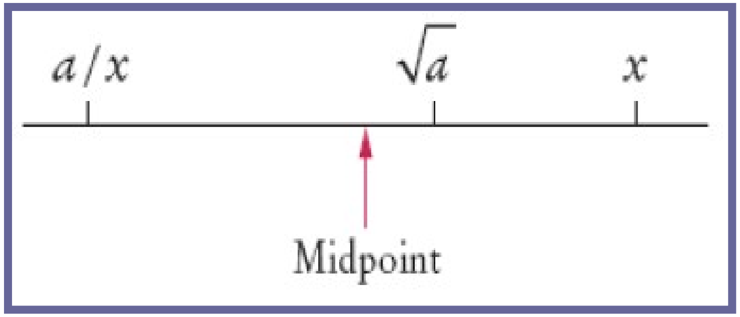
Testing and Debugging

- You don't test the complete program that you are developing; you test the classes in isolation
- For each test, you provide a simple class called a *test harness*
- Test harness feeds parameters to the methods being tested

Example: Setting Up Test Harnesses

- To compute the square root of *a* use a common algorithm:

1. Guess a value *x* that might be somewhat close to the desired square root (*x* = *a* is ok)
2. Actual square root lies between *x* and *a/x*
3. Take midpoint (*x* + *a*/*x*) / 2 as a better guess



4. Repeat the procedure. Stop when two successive approximations are very close to each other

Continued…

File RootApproximator.java

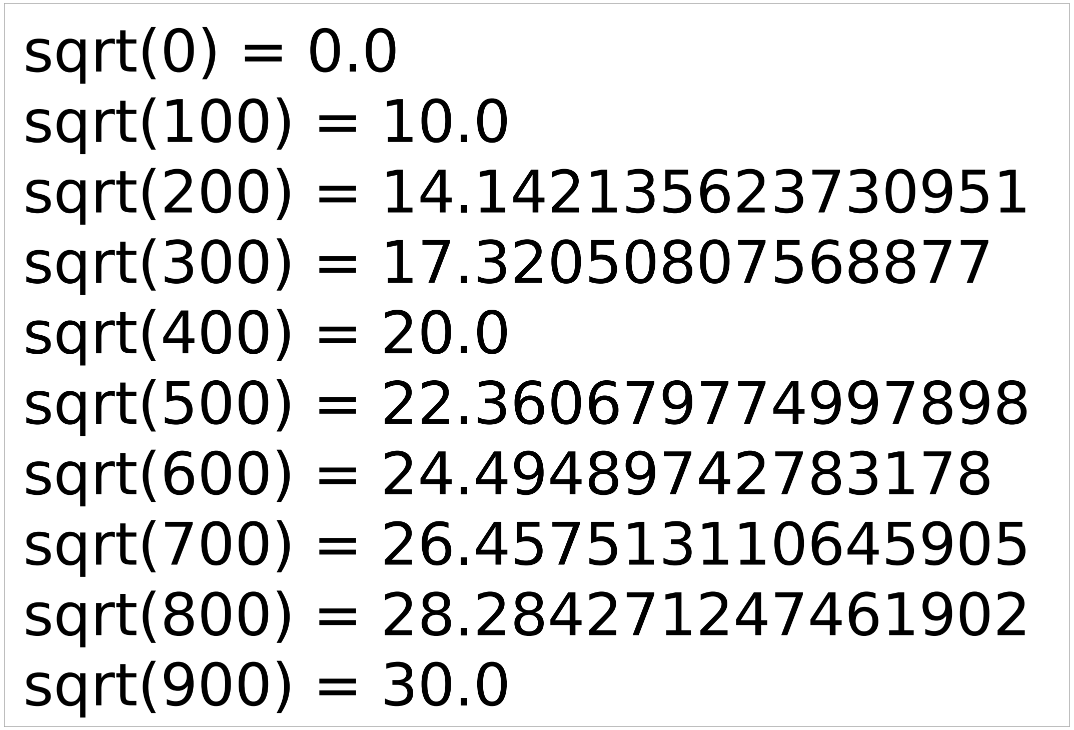
```
public class RootApproximator
{
   private static  final double EPSILON = 1E-12;

   public static double sqrt(double a)
   {
      double xold = 1;
      double x = a;
      while(!approxEqual(xold, x)){
         xold=x;
         x = (x+a/x)/2;
      }
      return x;
   }

   public static boolean approxEqual(double x, double y)
   {
      return Math.abs(x - y) <= EPSILON;
   }
}
```

File RootApproximator.java

```
public class RootApproximatorTester {
   public static void main(String[] args) {
       int i,m;
       for (i=0,m=1; m<=10; i+=100,m++)
          System.out.println("sqrt("+i+ ") = "+
                                RootApproximator.sqrt(i));
   }
}
```



```
sqrt(0) = 0.0
sqrt(100) = 10.0
sqrt(200) = 14.142135623730951
sqrt(300) = 17.32050807568877
sqrt(400) = 20.0
sqrt(500) = 22.360679774997898
sqrt(600) = 24.49489742783178
sqrt(700) = 26.457513110645905
sqrt(800) = 28.284271247461902
sqrt(900) = 30.0
```

Testing the Program

- Does the `RootApproximator` class work correctly for all inputs?
  It needs to be tested with more values
- If a problem is fixed and re-testing is needed, you would need to remember your inputs
- Solution: Write test harnesses that make it easy to repeat unit tests

Providing Test Input

There are various mechanisms for providing test cases

One mechanism is to hard-wire test inputs into the test harness

Simply execute the test harness whenever you fix a bug in the class that is being tested

File RootApproximatorHarness1.java

```java
/**
   This program computes square roots of selected input
      // values.
*/
public class RootApproximatorHarness1 {
   public static void main(String[] args){
      double[] testInputs = { 100, 4, 2, 1, 0.25, 0.01 };
      for (double x : testInputs) {
         double y = RootApproximator.sqrt(x);
         System.out.println("square root of " + x
               + " = " + y);
      }
   }
}
```

For few possible inputs, feasible to run through (representative) number of them with a loop

File RootApproximatorHarness2.java

```java
/**
   This program computes square roots of input values
   supplied by a loop.
*/
public class RootApproximatorHarness2 {
  public static void main(String[] args) {
    final double MIN = 1;
    final double MAX = 10;
    final double INCREMENT = 0.5;
    for (double x = MIN; x <= MAX; x = x + INCREMENT) {

      double y = RootApproximator.sqrt(x);
      System.out.println("square root of " + x
                                   + " = " + y);
    }
  }
}
```

- Test restricted to small subset of values
- Alternative: random generation of test cases

File RootApproximatorHarness3.java

```java
import java.util.Random;

/**
   This program computes square roots of random inputs.
*/
public class RootApproximatorHarness3 {
   public static void main(String[] args) {
      final double SAMPLES = 100;
      Random generator = new Random();
      for (int i = 1; i <= SAMPLES; i++) {
         // Generate random test value

         double x = 1000 * generator.nextDouble();
         double y = RootApproximator.sqrt(x);
         System.out.println("square root of " + x
               + " = " + y);
      }
   }
}
```

Random generation of test cases

Providing Test Input

- Selecting good test cases is an important skill for debugging programs
- Test all features of the methods that you are testing
- Test boundary test cases: test cases that are at the boundary of acceptable inputs
  0, for the `SquareRootApproximator`

Providing Test Input

- Programmers often make mistakes dealing with boundary conditions

  Division by zero, extracting characters from empty strings, and accessing null pointers

- Gather negative test cases: inputs that you expect program to reject

  Example: square root of -2. Test passes if harness terminates with assertion failure (if assertion checking is enabled)

Reading Test Inputs From a File

- More elegant to place test values in a file
- Input redirection:

```
java Program < data.txt
```

- Some IDEs do not support input redirection. Then, use command window (shell).
- Output redirection:

```
java Program > output.txt
```

File RootApproximatorHarness4.java

```java
import java.util.Scanner;
/**
   This program computes square roots of inputs supplied
   through System.in.
*/
public class RootApproximatorHarness4 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      boolean done = false;
      while (in.hasNextDouble()){
        double x = in.nextDouble();

         double y = RootApproximator.sqrt(x);

         System.out.println("square root of " + x
               + " = " + y);
      }
   }
}
```

Reading Test Inputs From a File

- File test.in:

```
1 100
2 4
3 2
4 1
5 0.25
6 0.01
```

Run the program:

```
java RootApproximatorHarness4 < test.in > test.out
```

Continued…

Test Case Evaluation

- How do you know whether the output is correct?
- Calculate correct values by hand
  E.g., for a payroll program, compute taxes manually
- Supply test inputs for which you know the answer
  E.g., square root of 4 is 2 and square root of 100 is 10

Continued…

Test Case Evaluation

- Verify that the output values fulfill certain properties
  E.g., square root squared = original value
- Use an *Oracle:* a slow but reliable method to compute a result for testing purposes
  E.g., use `Math.pow` to slower calculate $x^{1/2}$ (equivalent to the square root of x)

File RootApproximatorHarness5.java

```java
import java.util.Random;

public class RootApproximatorHarness5 {
   public static void main(String[] args) {
   final double SAMPLES = 100;
      int passcount = 0;
      int failcount = 0;
      Random generator = new Random();
      for (int i = 1; i <= SAMPLES; i++) {
         double x = 1000 * generator.nextDouble();
         double y = RootApproximator.sqrt(x);
         if (RootApproximator.approxEqual(y * y, x))
            passcount++;
         else
            failcount++;
      }
      System.out.println("Pass: " + passcount);
      System.out.println("Fail: " + failcount);
   }
}
```

Continued…

Regression Testing

- Save test cases
- Use saved test cases in subsequent versions
- A **test suite** is a set of tests for repeated testing
- **Cycling** : bug that is fixed but reappears in later versions
- **Regression testing**: repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Test Coverage

- **Black-box testing**: test functionality without consideration of internal structure of implementation
- **White-box testing**: take internal structure into account when designing tests
- **Test coverage**: measure of how many parts of a program have been tested
- Make sure that each part of your program is executed at least once by one test case

  E.g., make sure to execute each branch in at least one test case

Test Coverage

- Tip: write first test cases before program is written completely → gives insight into what program should do
- Modern programs can be challenging to test
  - Graphical user interfaces (use of mouse)
  - Network connections (delay and failures)
  - There are tools to automate testing in this scenarios
  - Basic principles of regression testing and complete coverage still hold

Program Trace

- Messages that show the path of execution

```
if (status == SINGLE)
{
   System.out.println("status is SINGLE");
. . .
}
. . .
```

Program Trace

- Drawback: Need to remove them when testing is complete, stick them back in when another error is found
- Solution: use the **`Logger`** class to turn off the trace messages without removing them from the program

Logging

- Logging messages can be deactivated when testing is complete
- Use Logger object
- Log a message

  You should import Logger first

```
import java.util.logging.*;
```

```
private static Logger theLogger =
Logger.getLogger(RootApproximatorTester.class.getName());


theLogger.info("status is SINGLE");
```

Continued…

Logging

1. Import the Logger

```
import java.util.logging.*;


Class ClassName{
```

2. Create a private static Logger as shown below

```
  private static Logger theLogger =
                 Logger.getLogger(ClassName.class.getName());
```

3. Instead of calling Println use Info() method of Logger

```
  theLogger.info("Message");
```

Continued…

Logging

- By default, logged messages are printed. Turn them off with

```
TheLogger.setLevel(Level.OFF);
```

- Logging can be a hassle (should not log too much nor too little)
- Some programmers prefer debugging (next section) to logging

Logging

- When tracing execution flow, the most important events are entering and exiting a method
- At the beginning of a method, print out the parameters:

```
public TaxReturn(double anIncome, int aStatus)
{
   TheLogger.info("Parameters: anIncome = " + anIncome
      + " aStatus = " + aStatus);
   . . .
}
```

Logging

- At the end of a method, print out the return value:

```
public double getTax()
{
   . . .
   TheLogger.info("Return value = " + tax);
   return tax;
}
```

Using a Debugger

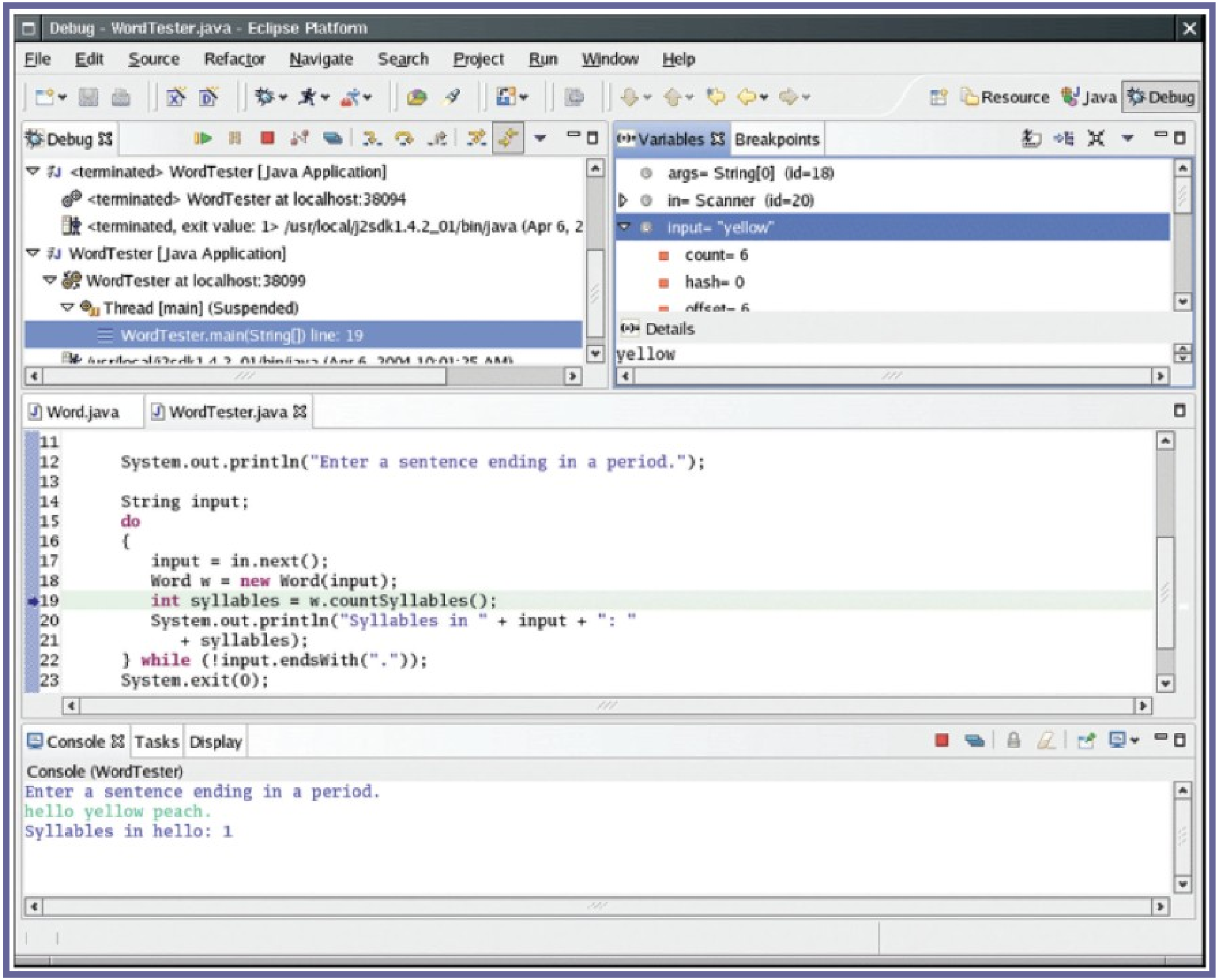
- Debugger : program to run your program and analyze its run-time behavior
- A debugger lets you stop and restart your program, see contents of variables, and step through it
- The larger your programs, the harder to debug them simply by logging

Continued…

Using a Debugger

- Debuggers can be part of your IDE (JGrasp, Eclipse, BlueJ) or separate programs (JSwat)
- Three key concepts:
  - Breakpoints
  - Single-stepping
  - Inspecting variables

The Debugger Stopping at a Breakpoint



```
<terminated> WordTester [Java Application]
    <terminated> WordTester at localhost:38094
    <terminated, exit value: 1> /usr/local/j2sdk1.4.2_01/bin/java (Apr 6, 2
WordTester [Java Application]
    WordTester at localhost:38099
        Thread [main] (Suspended)
            WordTester.main(String[]) line: 19
    /usr/local/j2sdk1.4.2_01/bin/java (Apr 6, 2004 10:01:25 AM)
```

```
args= String[0] (id=18)
in= Scanner (id=20)
input= "yellow"
    count= 6
    hash= 0
    offset= 6
```

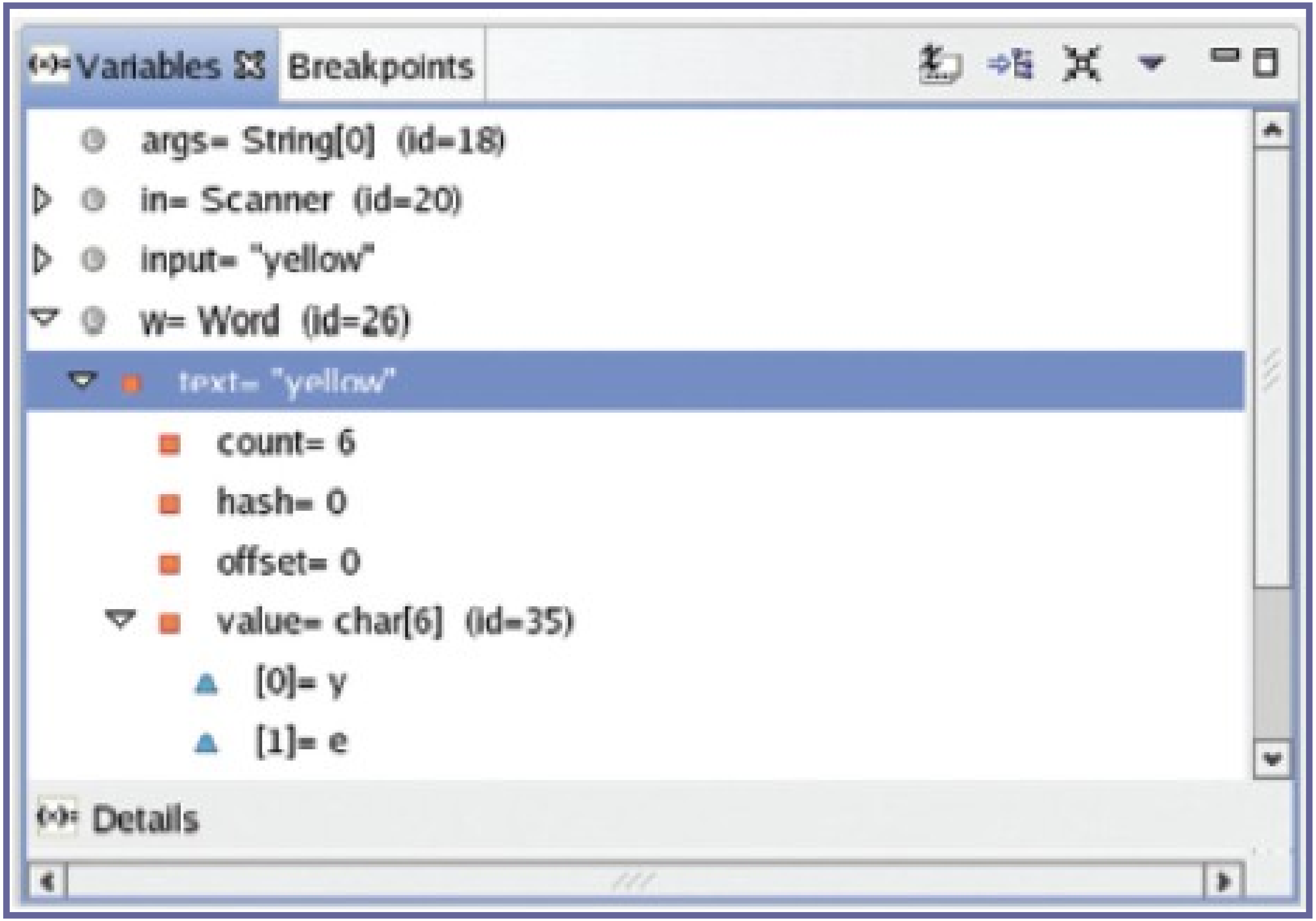
```
yellow
```

```java
11
12        System.out.println("Enter a sentence ending in a period.");
13
14        String input;
15        do
16        {
17           input = in.next();
18           Word w = new Word(input);
19           int syllables = w.countSyllables();
20           System.out.println("Syllables in " + input + ": "
21              + syllables);
22        } while (!input.endsWith("."));
23        System.exit(0);
```

Console (WordTester)

```
Enter a sentence ending in a period.
hello yellow peach.
Syllables in hello: 1
```

Inspecting Variables



```
args= String[0]  (id=18)
in= Scanner  (id=20)
input= "yellow"
w= Word  (id=26)
  text= "yellow"
    count= 6
    hash= 0
    offset= 0
    value= char[6]  (id=35)
      [0]= y
      [1]= e
```

Debugging

- Execution is suspended whenever a breakpoint is reached
- In a debugger, a program runs at full speed until it reaches a breakpoint
- When execution stops you can:
  - Inspect variables
  - Step through the program a line at a time
  - Or, continue running the program at full speed until it reaches the next breakpoint

Continued…

Debugging

- When program terminates, debugger stops as well
- Breakpoints stay active until you remove them
- Two variations of single-step command:
  - Step Over: skips method calls
  - Step Into: steps inside method calls

Another Error

- Fix the error
- Recompile
- Test again: