CHAPTER 3

Testing and Debugging

Chapter Objectives

- To understand different testing strategies
- To learn to test using the Junit test framework
- To learn to use test driven development
- To learn to use a debugger within a Java IDE

Types of Testing

Section 3.1

Types of Testing

- Testing is exercising a program under controlled conditions.
- More thorough testing increases the likelihood of finding defects.
- However, in a complex program, no amount of testing can guarantee the absence of defects.

Levels of Testing

- Unit testing
  - Tests the smallest testable pieces of the software
  - In OOD, this may be a class or a method.
- Integration testing
  - Tests interaction among units
  - If the unit is a method, this tests the interaction of methods within a class
  - More commonly, tests the interaction between several classes
- System testing
  - Tests the whole program in the context in which it will be used
- Acceptance testing
  - System testing designed to demonstrate that the program meets its functional requirements

Types of Testing

- Black-box testing
  - Tests the item based on its interfaces and functional requirements
  - Input values are varied over allowable ranges and outputs compared to independently calculated values.
  - Input values outside of allowed ranges are also tested to see if the unit responds according to specifications

Types of Testing (cont.)

- White-box testing
  - Tests the unit with knowledge of its internal structure
  - Attempts to exercise as many paths through the unit as possible
  - Statement coverage ensures that each statement is executed at least once
  - Branch coverage ensures that every choice at each branch is tested
  - Path coverage tests each path through a method

Example – testing all paths

- We want to test all of the paths of the following method.

```
public void testMethod(char a, char b) {
  if (a < 'M') {
    if (b < 'X') {
       System.out.println("path 1");
       ...
    } else {
       System.out.println("path 2");
       ...
    }
  } else {
    if (b < 'C') {
      System.out.println("path 3");
      ...
    } else {
      System.out.println("path 4");
      ...
    }
  }
}
```

Example (cont.)

- The following table shows possible input values to exercise all four possible paths:

a	b	Message
'A'	'A'	path1
'A'	'Z'	path2
'Z'	'A'	path3
'Z'	'Z'	path4

- These are the smallest and largest allowable values
- A more complete test should use additional valid combinations of values and with non-letter values

Preparations for Testing

- Planning for testing should begin early and include consideration of:
  - How will the program be tested?
  - When will it be tested?
  - By whom will it be tested?
  - What test data will be used?
- Early planning can help programmers prepare for testing as they write their code.
  - For instance, validating input data and throwing appropriate exceptions.

Testing Tips

- Document all class attributes and method parameters using comments.
- Trace execution by displaying each method name as it is entered.
- Display values of all input parameters as a method is entered. Also any class attributes used.
- After a method returns, display its return value and the values of any class attributes it modified.

Testing Tips (cont.)

- It is useful to include code like

```
if (TESTING) {
//code that you wish to "remove"
}
```

- Then you can add the following to your class when you want to enable testing

```
private static final boolean TESTING = true;
```

- And change it when you want to disable testing

```
private static final boolean TESTING = false;
```

Specifying the Tests

Section 3.2

Specifying the Tests – General Principles

- Black-box testing
  - Test all expected input values
  - Test unexpected input values
  - Specify anticipated results of each set of values tested

Specifying the Tests – General Principles (cont.)

- White-box testing
  - Exercise every branch of every if statement
  - Test switch statements for all valid selector values and some invalid values
  - Loops – test behavior if
    - The body is never executed
    - The body is executed once
    - The body is executed the maximum number of times
  - Assure that loops eventually will terminate

Boundary Conditions

- Boundary conditions are special cases which should be explicitly tested.
- For instance, in a method designed to find a specific value within an array, you would test cases where
  - The target is the first element in the array
  - The target is the last element in the array
  - The target is somewhere in the middle of the array
  - The target is not in the array

Boundary Conditions (cont.)

- More boundary conditions for a method that finds a specific target value in an array
  - There is more than one occurrence of the target value
  - The array has but one element and it is not the target
  - The array has but one element and it is the target
  - The array has no elements

Stubs and Drivers

Section 3.3

Stubs

- A stub is a replacement for a method not yet written
- The purpose of a stub is to allow early testing of components already written. An example follows:

```
/** Stub for method save.
  @pre the initial directory contents are read from a data file.
  @post Writes the directory contents back to a data file.
        The boolean flag modified is reset to false.
*/
public void save() {
    System.out.println("Stub for save has been called");
    modified = false;
}
```

Stubs (cont.)

- A stub should print an identifying message
- It could also print the values of its input parameters and any state variables that it may change
- Performing these operations allows the programmer to follow the flow of control within the client program is correct.

Preconditions and Postconditions

- Preconditions are the assumptions or constraints upon the input data for the method
- They should be documented in a comment using the @pre notation as in the preceding example
- Postconditions are any changes in state caused by the function.
- These should be documented for all void methods using the @post notation as in the preceding example.

Drivers

- A driver is a testing tool which consists of a program that creates any values and classes necessary to test a method.
- After calling the method, it displays the results of any output returned.
- Drivers are often conveniently executed as part of a test framework such as JUnit

The JUnit Test Framework

Section 3.4

The JUnit Test Framework

- A test harness is a program written to test a method or class
  - It provides known inputs for a series of tests (the test suite)
  - It compares the results with known results and reports whether the item under test passed or failed
- A test framework is a software product that facilitates writing and running test suites.
- We will demonstrate how to use a test framework named JUnit

Using JUnit

- Each test harness created in JUnit begins with two import statements:

```
import org.junit.Test;
import static org.junit.Assert.*
```

- These allow us to use JUnit's assert methods
- The assert methods allow us to specify pass/fail behavior for tests.
- They are summarized in Table 3.2 page ???

JUnit Example

- Design of a JUnit program to test the ArraySearch.search method.
- We wish to test the following:
  - The target is the first element in the array
  - The target is the last element in the array
  - The target is somewhere in the middle
  - The target is not in the array
  - There is more than one occurrence of the target and we find the first

JUnit Example (cont.)

- ArraySearch.search tests continues
  - The array has only one element and it is not the target
  - The array has only one element ant it is the target
  - The array has no elements
- The entire listing for the JUnit program is
- Listing 3.1 on page ???
- The common array used for all tests:

```
// Common array to search for most of the tests
    private final int[] x = {5, 12, 15, 4, 8, 12, 7};
```

JUnit Example (cont.)

- Testing the case where the target is the first element

```
@Test
    public void firstElementTest() {
        // Test for target as first element.
        assertEquals("5 not at position 0",
                0, ArraySearch.search(x, 5));
    }
```

- The “assertEquals” method specifies the message to print on failure, the expected result, and the function call. We expect a return value of 0 because 5 is indeed in the first array element.

JUnit Example (cont.)

- Testing the case where the target is the last element

```
@Test
    public void lastElementTest() {
        // Test for target as last element.
        assertEquals("7 not at position 6",
                6, ArraySearch.search(x, 7));
    }
```

- In this case, the target value was 7, and we expect to find it in location 6 (the last element of the array)

JUnit Example (cont.)

- Testing the case where the target is somewhere in the middle:

```
@Test
    public void inMiddleTest() {
        // Test for target somewhere in middle.
        assertEquals("4 is not found at position 3",
                3, ArraySearch.search(x, 4));
    }
```

- Here, the target value was 4 and we expect to find it in location 3.

JUnit Example (cont.)

- Testing the case where the target is not in the array

```
@Test
    public void notInArrayTest() {
        // Test for target not in array.
        assertEquals(-1, ArraySearch.search(x, -5));
    }
```

- Here, the target value was -5 and we expect a return value of -1 indicating “not found.”
- The first parameter to assertEquals is omitted which would result in a default failure message.

JUnit Example (cont.)

- Testing the case where the target is present in multiple locations, we find the first

```
@Test
    public void multipleOccurencesTest() {
        // Test for multiple occurrences of target.
        assertEquals(1, ArraySearch.search(x, 12));
    }
```

- Target = 12, which occurs at locations 1 and 5. We expect the program to return 1.

JUnit Example (cont.)

- Testing a 1 element array which does contain the target value

```
@Test
    public void oneElementArrayTestItemPresent() {
        // Test for 1-element array
        int[] y = {10};
        assertEquals(0, ArraySearch.search(y, 10));
    }
```

- We expect to find the 10 in location 0.

JUnit Example (cont.)

- Testing a 1 element array which does not contain the target value

```
@Test
    public void oneElementArrayTestItemAbsent() {
        // Test for 1-element array
        int[] y = {10};
        assertEquals(-1, ArraySearch.search(y, -10));
    }
```

- Y does not contain -10, so we expect a return value of -1 meaning “not found.”

JUnit Example (cont.)

- Testing with an empty array

```
@Test
    public void emptyArrayTest() {
        // Test for an empty array
        int[] y = new int[0];
        assertEquals(-1, ArraySearch.search(y, 10));
    }
```

- Y does not contain anything, so we expect a return value of -1 meaning “not found.”

JUnit Example (cont.)

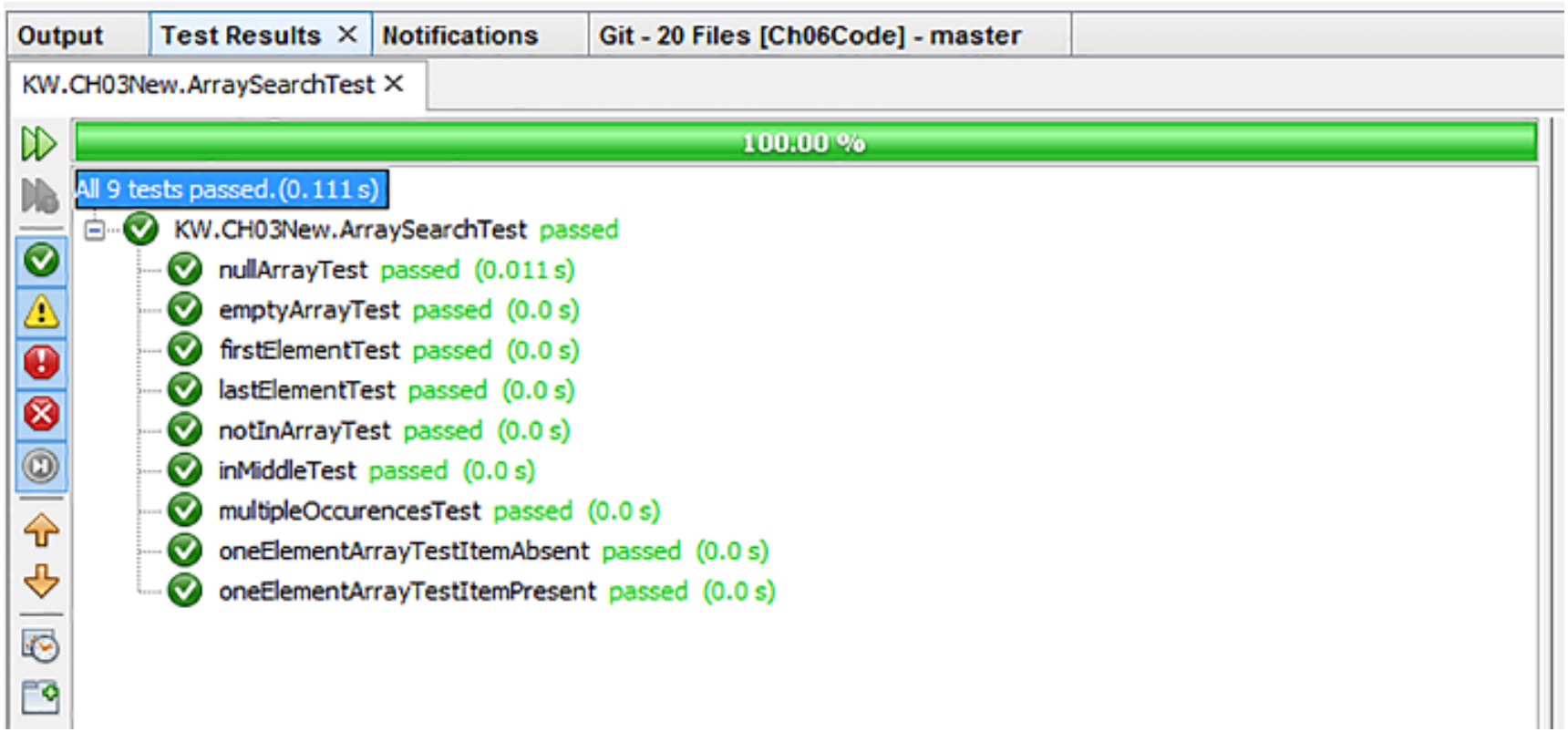
- Testing with a null pointer

```
@Test(expected=NullPointerException.class)
public void nullArrayTest() {
    int[] y = null;
    int i = ArraySearch.search(y, 10);
}
```

- Y is a null pointer. The @Test line says that the test is successful if we get a NullPointerException

JUnit Example (cont.)

- The results of the tests are shown below:



Test Driven Development

Section 3.5

Test Driven Development

- Test driven development involves writing tests and methods in parallel
  - Write a test case for a new feature
  - Run the test and note that it fails while other tests still pass
  - Make the minimum change necessary to make it pass
  - Revise the code to remove any duplication
  - Re-run the test to be sure it still passes

Case Study – ArraySearch.search

- Test list
  - The target element is not in the array
  - The target element is the first in the array
  - The target element is the last in the array
  - There is more than one occurrence, we find the first
  - The target is somewhere in the middle of the array
  - The array has only one element
  - The array has no elements

Case Study (cont.)

- Start with a stub

```
public class ArraySearch {
     /**
     * Search an array to find the first occurrence of a target
     * @param x Array to search
     * @param target Target to search for
     * @return The subscript of the first occurrence if found:
     * otherwise return -1
     * @throws NullPointerException if x is null
     */
    public static int search(int[] x, int target) {
        return Integer.MIN_VALUE;
    }
  }
```

Case Study (cont.)

- Create a test (cases 1 and 6 above)

```
/**
 * Test for ArraySearch class
 * @author Koffman & Wolfgang
 */
public class ArraySearchTest {
    @Test
    public void itemNotFirstElementInSingleElementArray() {
        int[] x = {5};
        assertEquals(-1, ArraySearch.search(x, 10));
    }
}
```

Case Study (cont.)

- The test fails (as we expected

```
Testcase: itemNotFirstElementInSingleElementArray:
        FAILED
expected:<-1> but was:<-2147483648>
```

- We make the minimum change to let it pass

```
public static int search(int[] x, int target) {
         return -1;              // target not found
    }
```

Case Study (cont.)

- Add a test for the target in the first location

```
@Test
 public void itemFirstElementInSingleElementArray() {
     int[] x = new int[]{5};
     assertEquals(0, ArraySearch.search(x, 5));
 }
```

- This again fails (as expected). We make a small change

```
public static int search(int[] x, int target) {
     if (x[0] == target) {
         return 0;          // target found at 0
     }
     return -1;             // target not found
 }
```

Case Study (cont.)

- Now the first two tests pass. We notice a possible improvement
- Returning 0 may not work when the array is larger, so we make another small change

```
public static int search(int[] x, int target) {
      int index = 0;
      if (x[index] == target)
          return index;          // target at 0
      return -1;                 // target not found
  }
```

- The first two tests still pass with the new code

Case Study (cont.)

- Creating a new test, for an array of size 2

```
@Test
 public void itemSecondItemInTwoElementArray() {
     int[] x = {10, 20};
     assertEquals(1, ArraySearch.search(x, 20));
 }
```

- This fails (as expected) because we never actually look at the second array element

Case Study (cont.)

- So we modify the code to make it pass

```
public static int search(int[] x, int target) {
      int index = 0;
      if (x[index] == target)
          return index;        // target at 0
      index = 1;
      if (x[index] == target)
          return index;        // target at 1
      return -1;               // target not found
  }
```

- This now fails with an array of size 1

Case Study (cont.)

- We fix this by checking if index is too large

```
public static int search(int[] x, int target) {
      int index = 0;
      if (x[index] == target)
          return index;        // target at 0
      index = 1;
      if (index < x.length)
          if (x[index] == target)
              return index;        // target at 1
      return -1;                   // target not found
  }
```

- And once again all tests pass

Case Study (cont.)

- Now we make a test for a still larger array

```
@Test
 public void itemLastInMultiElementArray() {
     int[] x = new int[]{5, 10, 15};
     assertEquals(2, ArraySearch.search(x, 15));
 }
```

- This fails if the target is anywhere beyond location 1

Case Study (cont.)

- So we add code to check the rest of the locations

```
public static int search(int[] x, int target) {
        int index = 0;
        if (x[index] == target)
            return index;           // target at 0
        index = 1;
        while (index < x.length) {
           if (x[index] == target)
               return index;    // target at index
           index++;
        }
        return -1;              // target not found
    }
```

- This now passes all of our tests. However…

Case Study (cont.)

- We add a test to check that it returns -1 for an empty array

```
@Test
    public void itemNotInEmptyArray() {
        int[] x = new int[0];
        assertEquals(-1, ArraySearch.search(x, 5));
    }
```

- This fails, because our test for index < x.length comes too late.

Case Study (cont.)

- Examining our code, we note that we can fix this error and make the code shorter at the same time by eliminating the special case test location 0.

```
} public static int search(int[] x, int target) {
        int index = 0;
        while (index < x.length) {
           if (x[index] == target)
               return index;       // target at index
           index++;
        }
        return -1;                 // target not found
    }
```

- This version now passes all of our tests

Testing Interactive Programs

Section 3.6

Interactive Programs and JUnit

- The text contains a program that solicits the user for an integer in a specific range. We want to use JUnit to test such a program.
- One advantage of a test framework is that it is automated and all of the input is specified in advance. How can we apply this to a program that demands user input?

Interactive Programs and Junit (cont.)

- The solution lies in the use of ByteArrayInputStream and ByteArrayOutputStream.
- ByteArrayInputStream is a form of InputStream that consists of a fixed array of bytes.

Interactive Programs and Junit (cont.)

- The following test code provides the string "3" to the program being tested just as if it had been typed by a human user

```
@Test
    public void testForNormalInput() {
        ByteArrayInputStream testIn =
            new ByteArrayInputStream("3".getBytes());
        System.setIn(testIn);
        int n = MyInput.readInt("Enter weight", 2, 5);
        assertEquals(n, 3);
    }
```

Interactive Programs and Junit (cont.)

- To capture the prompt, we need to create a ByteArrayOutputStream that can be filled in by the programs System.out.print statements

```
@Test
    public void testForNormalInput() {
        ByteArrayInputStream testIn =
            new ByteArrayInputStream("3".getBytes());
        System.setIn(testIn);
        int n = MyInput.readInt("Enter weight", 2, 5);
        assertEquals(n, 3);
    }
```

Interactive Programs and Junit (cont.)

- This does the job

```
@Test
public void testThatPromptIsCorrectForNormalInput() {
    ByteArrayInputStream testIn =
            new ByteArrayInputStream("3".getBytes());
    System.setIn(testIn);
    ByteArrayOutputStream testOut = new ByteArrayOutputStream();
    System.setOut(new PrintStream(testOut));
    int n = MyInput.readInt("Enter weight", 2, 5);
    assertEquals(n, 3);
    String displayedPrompt = testOut.toString();
    String expectedPrompt = "Enter weight" +
            "\nEnter an integer between 2 and 5" + NL;
    assertEquals(expectedPrompt, displayedPrompt);
}
```

Interactive Programs and Junit (cont.)

- The line that reads

```
String expectedPrompt = "Enter weight" +
                        "\nEnter an integer between 2 and 5" + NL;
```

- Assumes that NL has been given a meaning. NL is meant to represent the newline character and this is system specific. To find what it is for your system you can add

```
private static final String NL =
        System.getProperty("line.separator");
```

Debugging

Section 3.7

Debugging

- Debugging is like detective work
- You must search for clues in the output information your program gives you.
- Sometimes, you need to ask, temporarily for more output than your program is currently providing

Debugging (cont.)

- The loop below does not terminate when the user enters the sentinel string (“***”).

```
public static String getSentence() {
        Scanner in = new Scanner(System.in);
        StringBuilder stb = new StringBuilder();
        int count = 0;
        while (count < 10) {
            System.out.println("Enter a word or *** to
quit");
            String word = in.next();
            if (word == "***") break;
            stb.append(word);
        }
```

Debugging (cont.)

- To better understand the problem, you add the following line of code as the first statement of the loop body. This will display the progressive sequence of words entered.

```
System.out.println("!!! Next word is " + word + ", count
is " + count);
```

Debugging (cont.)

- Running this will show you that *** does appear, but does not trigger the loop exit. This suggests that there is something wrong with the loop condititon.
- The problem is that the condition word == “***” is comparing addresses. The correct while condition is

```
while (word != null && !word.equals("***") && count < 10)
```

Using a Debugger

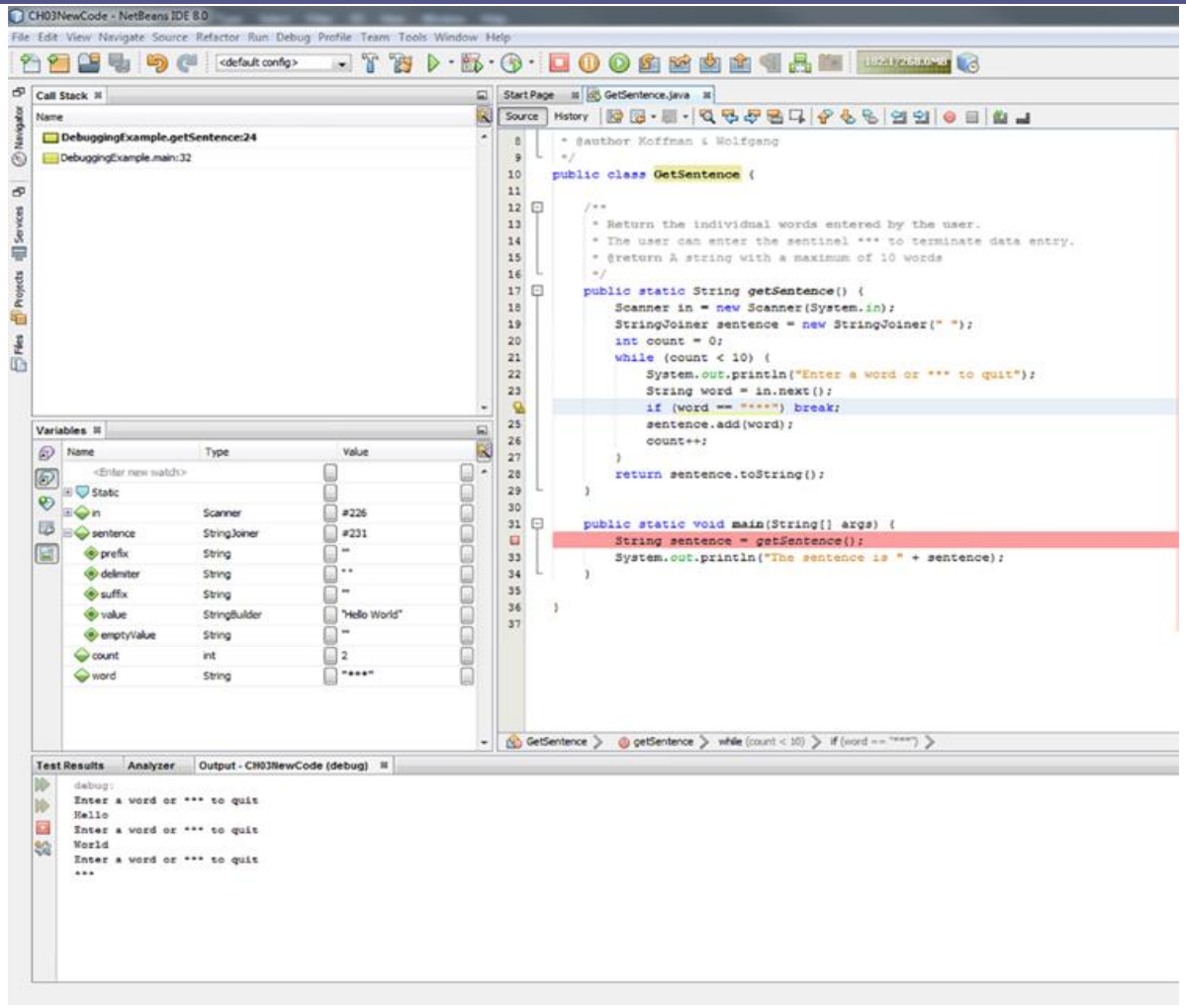
- Most IDE's contain debugging tools
- These allow you to execute your code incrementally, to set points where execution will stop and allow you to look at the contents of various memory cells.

Using a Debugger

- In Netbeans, for instance before running your program you can set a “breakpoint,” a line where your program will pause.
- You do this by clicking on the vertical bar to the left of the statement where you want to pause.

- The next slide contains a screenshot of a Netbeans showing the values of variables while paused at a breakpoint

Using a Debugger (cont.)



Using a Debugger (cont.)

- The use of a debugger in an IDE such as Netbeans can substantially speed up the process of finding bugs over the use of temporary print statements.
- However, no debugging tool can replace a thoughtful, logical approach to zeroing in on coding errors.