



Statistiche prima prova

- 1° turno
 - Prenotati: 41
 - Presenti: 31
 - **Assenti: 10**
- 2° turno
 - Prenotati: 41
 - Presenti: 33
 - **Assenti: 8**
- 3° turno
 - Prenotati: 16
 - Presenti: 10
 - **Assenti: 6**
- Totale **Assenti: 24**
- **2 gruppi bastavano**
- **Conclusione degli studenti del 3° gruppo e del docente in aula**
- **MALEDETTI!**



Correzione di errori (debugging)



Program Trace

- Messaggi che mostrano un cammino di esecuzione:

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

- Stampare il contenuto dello stack di esecuzione:

```
Throwable t = new Throwable();
t.printStackTrace(System.out);
```



Logging

- Svantaggio uso *program tracing*: bisogna rimuovere i messaggi una volta che il testing è completo e reinserirli se viene individuato un nuovo errore
- Alternativa: usare la classe **Logger** per zittire le tracce di messaggi senza rimuoverli
- Invece di stampare sul flusso **System.out** si può usare l'oggetto di *logging* globale

```
Logger logger = Logger.getLogger("nome-package");
```



Logging

- Il logger appena creato scrive in memoria. Per farlo scrivere su un file bisogna appoggiarsi ad un altro tipo di classe, detta Handler. Ci sono vari tipi di Handler standard:
 - StreamHandler: Scrive su un OutputStream.
 - ConsoleHandler: Scrive su System.err
 - FileHandler: Scrive su un file o vari file a rotazione.
 - SocketHandler: Scrive su una porta TCP remota.
- Per scrivere su un file utilizzeremo la classe **FileHandler**

```
Logger logger = Logger.getLogger("nome-package");  
static FileHandler fh= new FileHandler("nomefile");  
logger.addHandler(fh);
```



Logging

- “Loggare” un messaggio

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

- Il logging dei messaggi può essere disattivato quando il testing è completo

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

- Per usare **Logger** e **Level** bisogna importarli dal pacchetto: **java.util.logging**



Logging

- Nel tracciare il flusso di esecuzione di un programma, gli eventi più importanti sono l'entrata e l'uscita da un metodo
- Un utile informazione da visualizzare è il valore dei parametri all'ingresso:

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



Logging

- Alla fine del metodo può essere utile stampare il valore restituito:

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




Assertzioni

- Per controllare condizioni di errore è possibile usare anche le asserzioni (già studiate)

Esempio:

```
assert amount >= 0;  
balance = balance + amount;
```

(programma si interrompe con segnalazione di un **AssertError** se l'asserzione non è verificata)

- Le asserzioni sono disattivabili una volta che il programma supera il collaudo
- Si usano principalmente per controllare precondizioni



Debugger

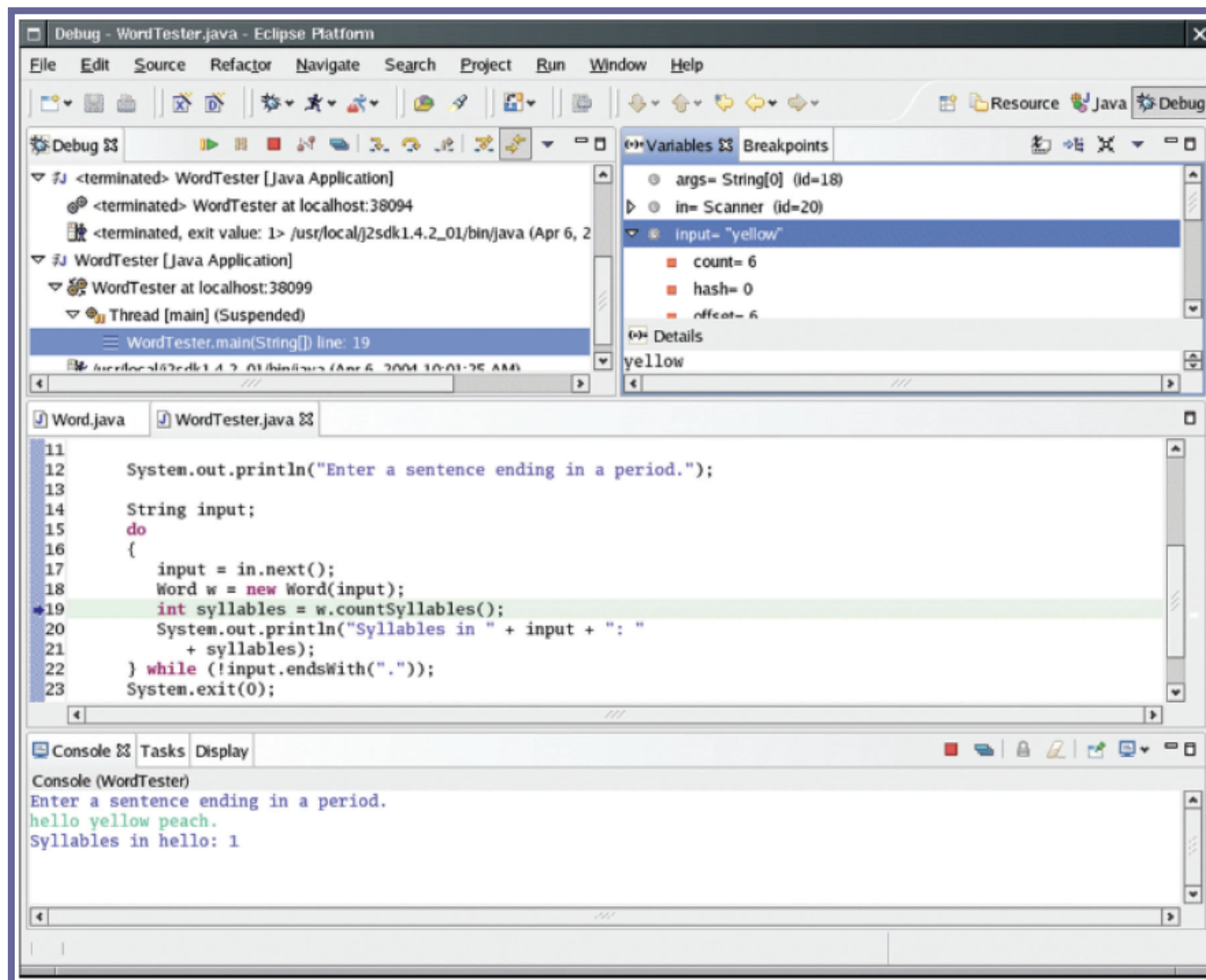
- Per programmi molto grandi semplicemente il logging non è praticabile
- La maggior parte dei programmatori usano un debugger per individuare e correggere un errore
- Debugger = un programma che esegue il programma sotto collaudo e analizza il suo comportamento a run-time
- Un debugger permette di arrestare e far ripartire l'esecuzione del programma (usando i *breakpoint*), visualizzare il contenuto delle variabili e eseguire il programma un passo alla volta (*single step*)



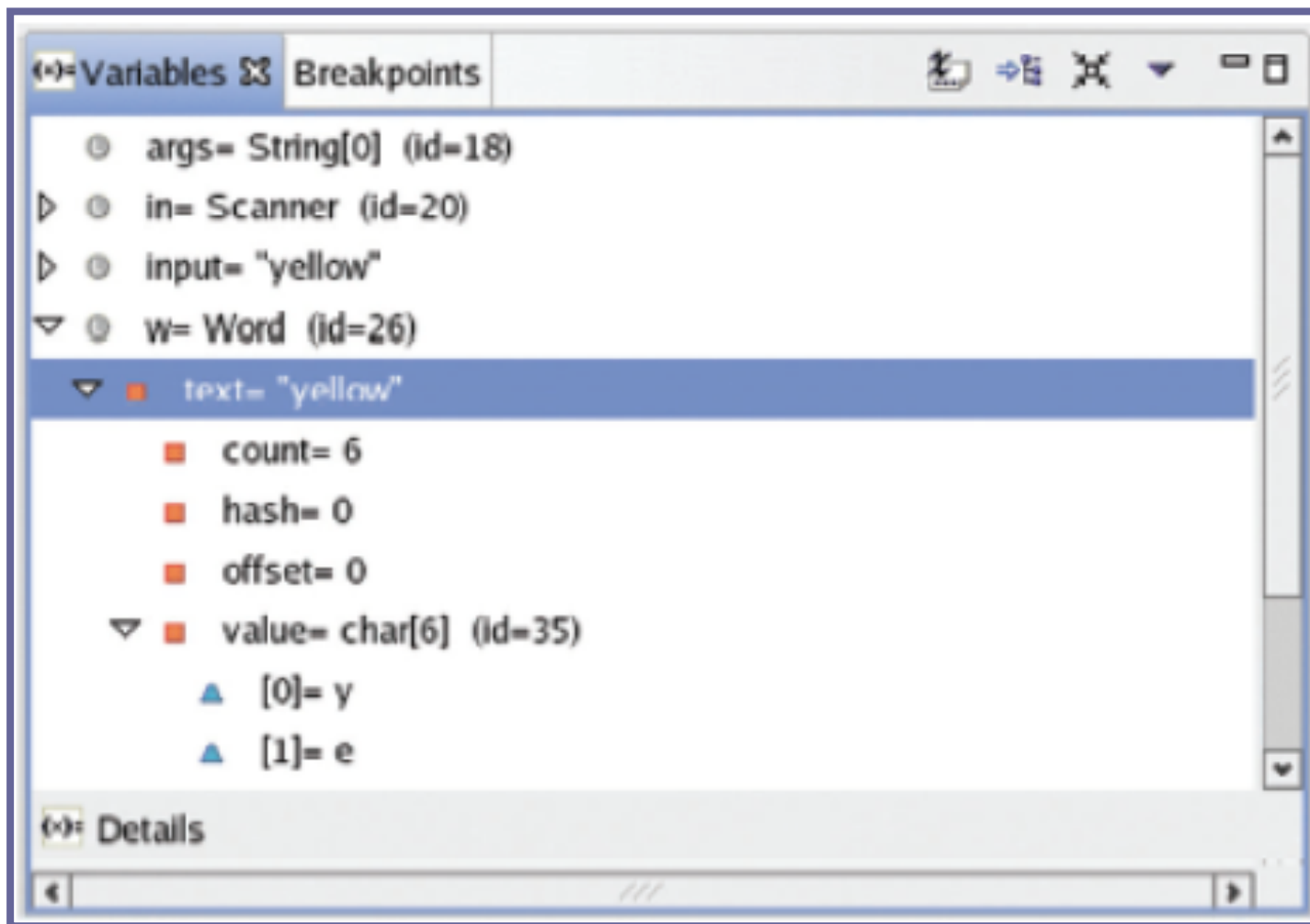
Debugger

- Debuggers sono sia parte di un IDE (come Eclipse, NetBeans, IntelliJ IDEA) o programmi a parte (JSwat)
- Concetti chiave:
 - Breakpoints
 - Single-stepping
 - Ispezione delle variabili

Stop a un Breakpoint



Ispezionare le variabili





Debugging

- L' esecuzione è sospesa ogni volta che viene raggiunto un breakpoint
 - Tra due breakpoint il programma viene eseguito normalmente
- Quando l' esecuzione si arresta si può:
 - Ispezionare le variabili
 - Eseguire il programma una linea alla volta
 - Oppure eseguire il programma in maniera normale finché non raggiunge il prossimo breakpoint



Debugging

- Quando il programma termina, anche il debugger termina
- I breakpoint restano attivi finché non vengono rimossi
- Ci sono due varianti del comando single-step:
 - Step Over: salta le chiamate a metodi
 - Step Into: esegue le chiamate a metodi



Esempio Single-Step

- Linea corrente:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " + syllables);
```

- Quando si sceglie “Step over” la chiamata, si passa alla riga successiva:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " + syllables);
```




Esempio Single-Step

- Se si sceglie “Step into” la chiamata a metodo, si passa alla prima riga del metodo **countSyllables** method

```
public int countSyllables()  
{  
    int count = 0;  
    int end = text.length() - 1;  
    . . .  
}
```



Un esempio di una sessione di Debugging

- La classe **Word** conta le sillabe in una parola
- Ogni gruppo di lettere in (a, e, i, o, u, y) adiacenti forma una sillaba
- Tuttavia una “e” alla fine di una parola non conta come una sillaba
- Ad esempio:
 - “real”: “ea” forma una sillaba
 - “regal”; “e..a” formano due sillabe
- Il costruttore di **Word** rimuove i caratteri non lettere all’inizio e alla fine della parola



File Word.java

```
01: /**
02:     This class describes words in a document.
03: */
04: public class Word
05: {
06:     /**
07:         Constructs a word by removing leading and trailing non-
08:         letter characters, such as punctuation marks.
09:         @param s the input string
10:     */
11:     public Word(String s)
12:     {
13:         int i = 0;
14:         while (i < s.length() && !Character.isLetter(s.charAt(i)))
15:             i++;
16:         int j = s.length() - 1;
17:         while (j > i && !Character.isLetter(s.charAt(j)))
18:             j--;
```



File Word.java

```
19:         text = s.substring(i, j);
20:     }
21:
22:     /**
23:         Returns the text of the word, after removal of the
24:         leading and trailing non-letter characters.
25:         @return the text of the word
26:     */
27:     public String getText()
28:     {
29:         return text;
30:     }
31:
32:     /**
33:         Counts the syllables in the word.
34:         @return the syllable count
35:     */
```



File Word.java

```
36:     public int countSyllables()
37:     {
38:         int count = 0;
39:         int end = text.length() - 1;
40:         if (end < 0) return 0; // The empty string has no
           // syllables
41:
42:         // An e at the end of the word doesn't count as a vowel
43:         char ch = Character.toLowerCase(text.charAt(end));
44:         if (ch == 'e') end--;
45:
46:         boolean insideVowelGroup = false;
47:         for (int i = 0; i <= end; i++)
48:         {
49:             ch = Character.toLowerCase(text.charAt(i));
50:             String vowels = "aeiouy";
51:             if (vowels.indexOf(ch) >= 0)
52:             {
```



File Word.java

```
53:                // ch is a vowel
54:                if (!insideVowelGroup)
55:                {
56:                    // Start of new vowel group
57:                    count++;
58:                    insideVowelGroup = true;
59:                }
60:            }
61:        }
62:
63:        // Every word has at least one syllable
64:        if (count == 0)
65:            count = 1;
66:
67:        return count;
68:    }
69:
70:    private String text;
71: }
```



File WordTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:     This program tests the countSyllables method of the Word
        // class.
05: */
06: public class WordTester
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.println("Enter a sentence ending in a
            period.");
13:
14:         String input;
15:         do
16:         {
```



File WordTester.java

```
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:     }
23:     while (!input.endsWith("."));
24: }
25: }
```




Facciamo il debugging del programma

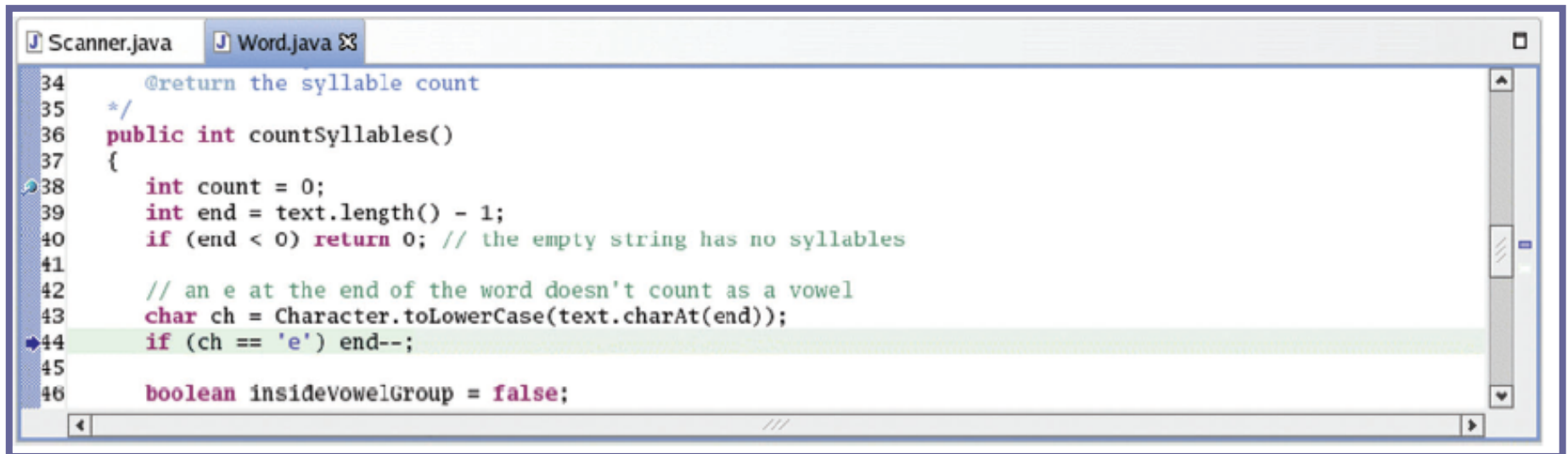
- Output erroneo (per input "hello yellow peach"):

```
Syllables in hello: 1  
Syllables in yellow: 1  
Syllables in peach: 1
```

- Metti un breakpoint nella prima riga di `countSyllables` della classe `Word`
- Esegui il programma nel debugger fornendo l'input. Il programma si ferma al breakpoint
- Il metodo testa se l'ultima lettera è una 'e'

Debugging

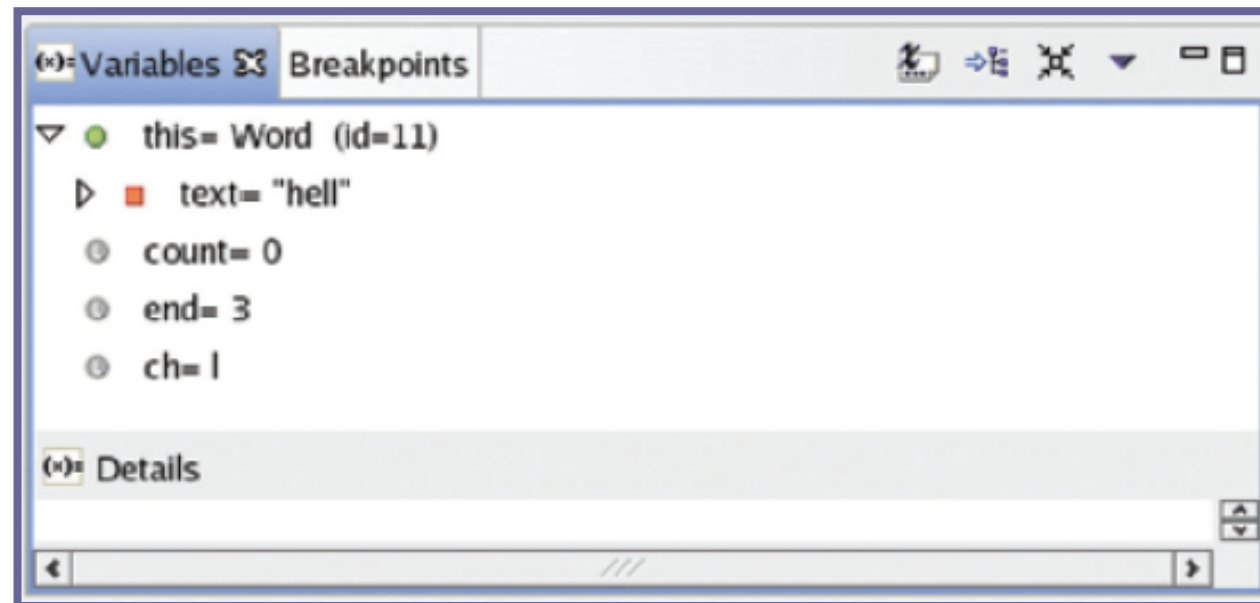
- Verifica se è vero: muovi passo-passo fino all'istruzione if-then che esegue il check



```
34  @return the syllable count
35  */
36  public int countSyllables()
37  {
38      int count = 0;
39      int end = text.length() - 1;
40      if (end < 0) return 0; // the empty string has no syllables
41
42      // an e at the end of the word doesn't count as a vowel
43      char ch = Character.toLowerCase(text.charAt(end));
44      if (ch == 'e') end--;
45
46      boolean insideVowelGroup = false;
```

Altri problemi rilevati

- Ispeziona la variabile `ch`
 - Dovrebbe contenere la lettera finale ma contiene 'l'
- `end` vale 3 e non 4
- `text` contiene "hell", e non "hello"
- Abbiamo scoperto perchè `countSyllables` restituisce 1





Debugging il costruttore di `Word`

- Il problema è nel costruttore di `Word`
- Si inserisce un `break` alla fine del secondo loop nel costruttore
- Si fornisce `"hello"` come input
- Ispeziona i valori di `i` e `j`
- Valgono 0 e 4
 - giusto perchè l'input consiste di sole lettere
- Perchè allora `text` vale `"hell"`?



Debugging il costruttore di Word

- Individuato un errore: il secondo parametro di **substring** è la prima posizione *non* inclusa
- `text = substring(i, j);`
dovrebbe essere
`text = substring(i, j + 1);`



Un altro Errore

- Si corregge l'errore trovato
- Si ricompila
- Si testa di nuovo e si ottiene come output:

```
Syllables in hello: 1  
Syllables in yellow: 1  
Syllables in peach: 1
```

- Il valore non è ancora corretto



Un altro Error

- Fai partire il debugger
- Cancella tutti i vecchi breakpoint e aggiungi un breakpoint all'inizio del metodo `countSyllables`
- Dai "hello" come input
- Quindi continua un passo alla volta (Single step) fino a giungere al ciclo "for"



Debugging CountSyllables

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = Character.toLowerCase(text.charAt(i));
    if ("aeiouy".indexOf(ch) >= 0)
    {
        // ch is a vowel
        if (!insideVowelGroup)
        {
            // Start of new vowel group
            count++;
            insideVowelGroup = true;
        }
    }
}
```




Debugging `CountSyllables`

- Prima iterazione ('h'): salta il test per vocali
- Seconda iterazione ('e'): passa il test e incrementa `count`
- Terza iterazione ('l'): salta il test
- Quarta iterazione ('l'): salta il test
- Quinta iterazione ('o'): passa il test, ma il secondo `if` è saltato, e `count` non è incrementato



Correzione dell' errore

- `insideVowelGroup` non è stata più settata a “false”
- Si corregge

```
if ("aeiouy".indexOf(ch) >= 0)
{
    . . .
}
else insideVowelGroup = false;
```



Ripetiamo il procedimento

- Riesegui il test: risultato corretto per tutti gli input forniti nel nostro test campione

```
Syllables in hello: 2  
Syllables in yellow: 2  
Syllables in peach: 1
```

- Si può dire ora che il programma è funziona correttamente? Il debugger non può dare questo tipo di risposte