Programming Languages and Techniques (CIS120)

Lecture 31

I/O & Histogram Demo

Chapter 28

Exceptions (recap)

Exceptions

- Exceptions are just objects that affect control flow:
- Raise an exception with:
  throw new ExceptionType();
  - aborts the current execution context (workspace)
  - "unwinds" the stack, searching for a matching catch block

- Handle exceptions using try/catch:
  try { /* code */ }
  catch (ExceptionType e) { /* handler */ }
  - runs code
  - if code raises an exception that is a subtype of ExceptionType, intercept its stack unwinding and run the handler

Finally

```
try {
    ...
} catch (Exn1 e1) {
   ...
} catch (Exn2 e2) {
   ...
} finally {
   ...
}
```

- A `finally` clause of a try/catch/finally statement *always* gets run, regardless of whether there is no exception, a propagated exception, or a caught exception.

Using Finally

- `Finally` is often used for releasing resources that might have been held/created by the `try` block:

```
public void doSomeIO (String file) {
  FileReader r = null;
  try {
    r = new FileReader(file);
    … // do some IO
  } catch (FileNotFoundException e) {
    … // handle the absent file
  } catch (IOException e) {
    … // handle other IO problems
  } finally {
    if (r != null) {    // don't forget null check!
      try { r.close(); } catch (IOException e) {…}
    }
  }
}
```

Using Finally

```
class C {
      public void foo() {
        this.bar();
        System.out.println("here in foo");
      }
      public void bar() {
        try {
          this.baz();
        } catch (Exception e) {
             System.out.println("caught");
        } finally { System.out.println("finally"); }
        System.out.println("here in bar");
      }
      public void baz() {
        throw new RuntimeException();
      }
    }
```

What happens if we do `(new C()).foo()` ?

1. Program prints only "finally"
2. Program prints "here in bar", then "here in foo", then "finally"
3. Program prints "finally", then "caught", then "here in foo"
4. Program prints "caught", then "finally", then "here in bar", then "here in foo"

Answer: 4

Using Finally

```
class C {
      public void foo() {
        this.bar();
        System.out.println("here in foo");
      }
      public void bar() {
        try {
          this.baz();
        } catch (Exception e) {
             System.out.println("caught");
        } finally { System.out.println("finally"); }
        System.out.println("here in bar");
      }
      public void baz() {
        throw new RuntimeException();
      }
    }
```

Good Style for Exceptions

- In Java, exceptions should be used to capture *exceptional circumstances*
  - Try/catch/throw incur performance costs and complicate reasoning about the program, don't use them when better solutions exist

- *Re-use existing exception types* when they are meaningful to the situation
  - e.g. use NoSuchElementException when implementing a container

- Define your own subclasses of Exception if doing so can convey useful information to possible callers that can handle the exception.

Good Style for Exceptions

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - e.g. when implementing `WordScanner` (in upcoming lectures), we catch `IOException` and throw `NoSuchElementException` in the `next` method.

- Catch exceptions as near to the source of failure as makes sense
  - i.e. where you have the information to deal with the exception

- Catch exceptions with as much precision as you can

  BAD: `try {…} catch (Exception e) {…}`
  BETTER: `try {…} catch (IOException e) {…}`

java.io

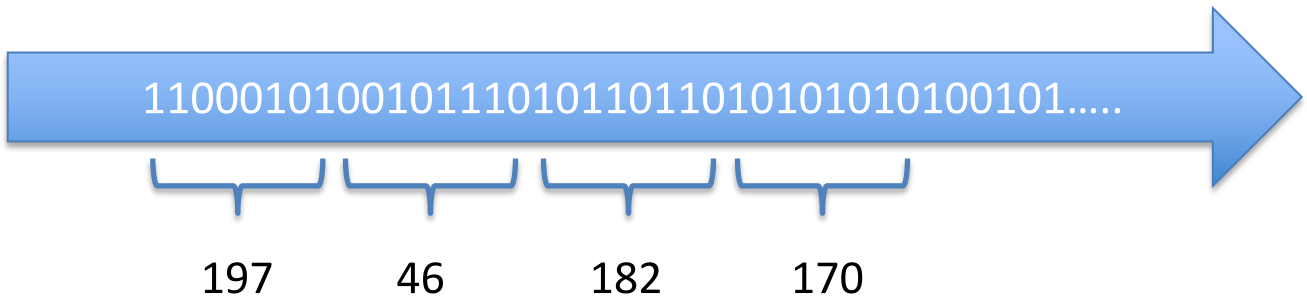
InputStream, OutputStream,
FileInputStream,PrintStream, Reader, Writer

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.

Low-level (Binary) Streams

- At the lowest level, a stream is a sequence of binary numbers



- The simplest IO classes break up the sequence into **8-bit** chunks, called *bytes*. Each byte corresponds to an integer in the range 0 – 255.

InputStream and OutputStream

- Abstract classes that provide basic operations for the Stream class hierarchy:

```
int read ();          // Reads the next byte of data
void write (int b); // Writes the byte b to the output
```

- These operations read/write `int` values that represent *bytes*

  range `0-255` represents a byte value

  `-1` represents "no more data" (when returned from read)

- java.io provides subclasses for various sources/sinks of data:

  files, audio devices, strings, byte arrays, serialized objects

- Subclasses also provide rich functionality:

  encoding, buffering, formatting, filtering

Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[][] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

BufferedInputStream

- Reading one byte at a time can be slow!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.

  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program

- A `BufferedInputStream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

  disk -> operating system ->>>> JVM -> program
  JVM -> program
  JVM -> program
  JVM -> program

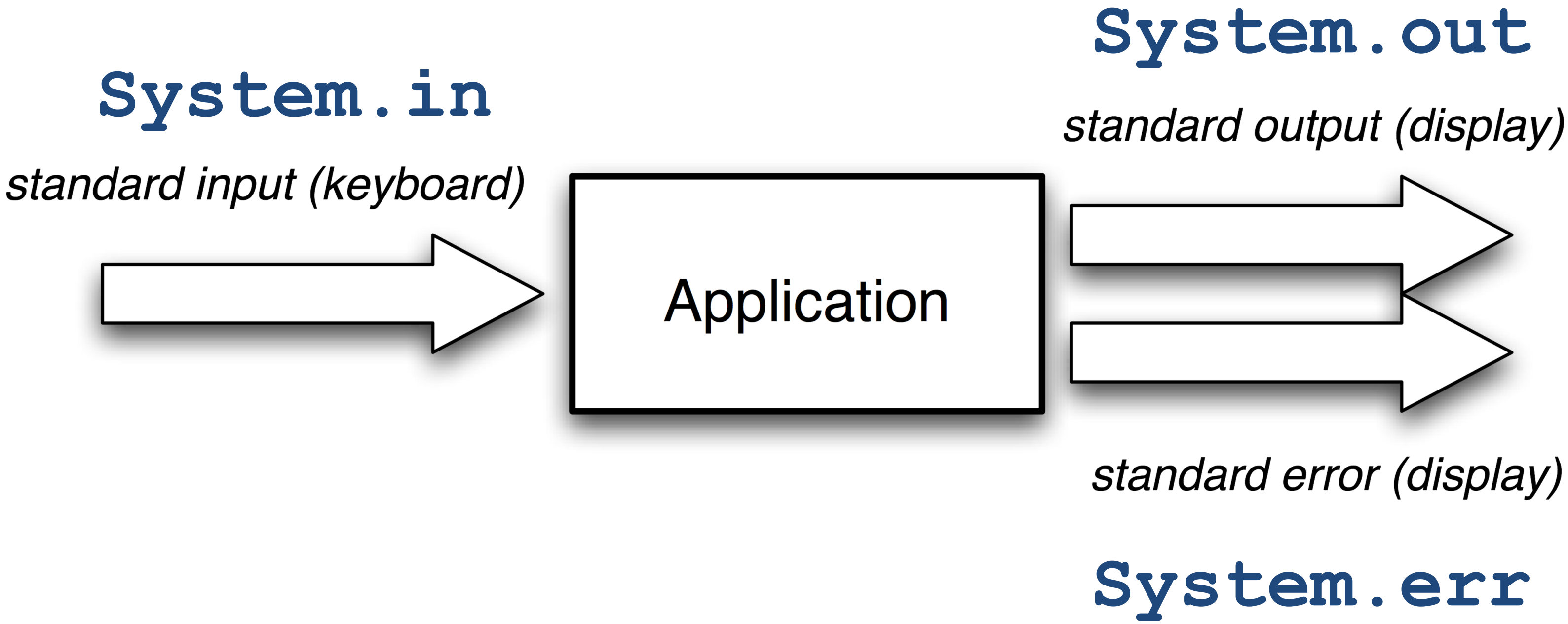
Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[][] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
         fin.close();
         throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

The Standard Java Streams

java.lang.System provides an `InputStream` and two standard `PrintStream` objects for doing console I/O.



Note that `System.in`, for example, is a *static member* of the class System – this means that the field "`in`" is associated with the *class*, not an *instance* of the class. Recall that static members in Java act like global variables.

`PrintStream` Methods

`PrintStream` adds buffering and binary-conversion methods to `OutputStream`

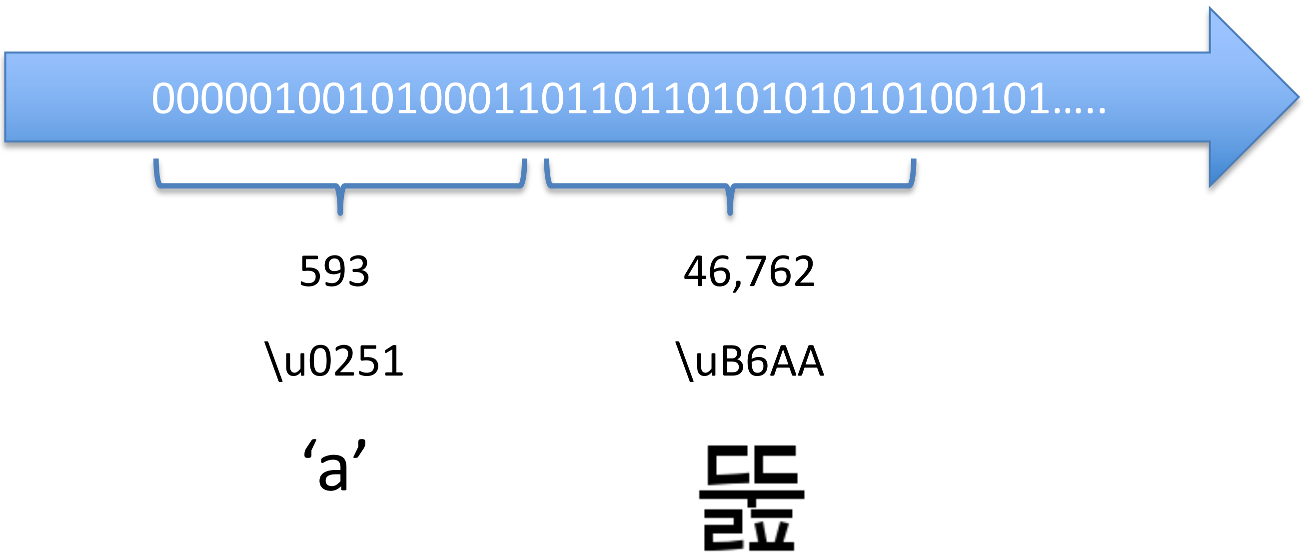
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

void print(String s);    // write s without terminating the line
                         (output may not appear until the stream is flushed)
void flush();            // actually output characters waiting to be sent
```

- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method's call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to "glue together" the right stream processing routines

Character based IO

A character stream is a sequence of **16-bit** binary numbers



The character-based IO classes break up the sequence into 16-bit chunks, of type char. Each character corresponds to a letter (specified by a *character encoding*).

`Reader` and `Writer`

- Similar to the `InputStream` and `OutputStream` classes, including:

```
int read ();          // Reads the next character
void write (int b); // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - read returns an integer in the range 0 to 65535 (i.e. 16 bits)
  - value `-1` represents "no more data" (when returned from read)
  - requires an "encoding" (e.g. UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, java.io provides many subclasses of Reader and Writer
  - e.g. FileReader / FileWriter
  - use classes these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - So wrap in an InputStreamReader / PrintWriter if you need unicode console I/O

Design Example: Histogram.java

A design exercise using java.io and the generic collection libraries

(SEE COURSE NOTES)

Problem Statement

Write a program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of “word: freq” pairs (one per line).

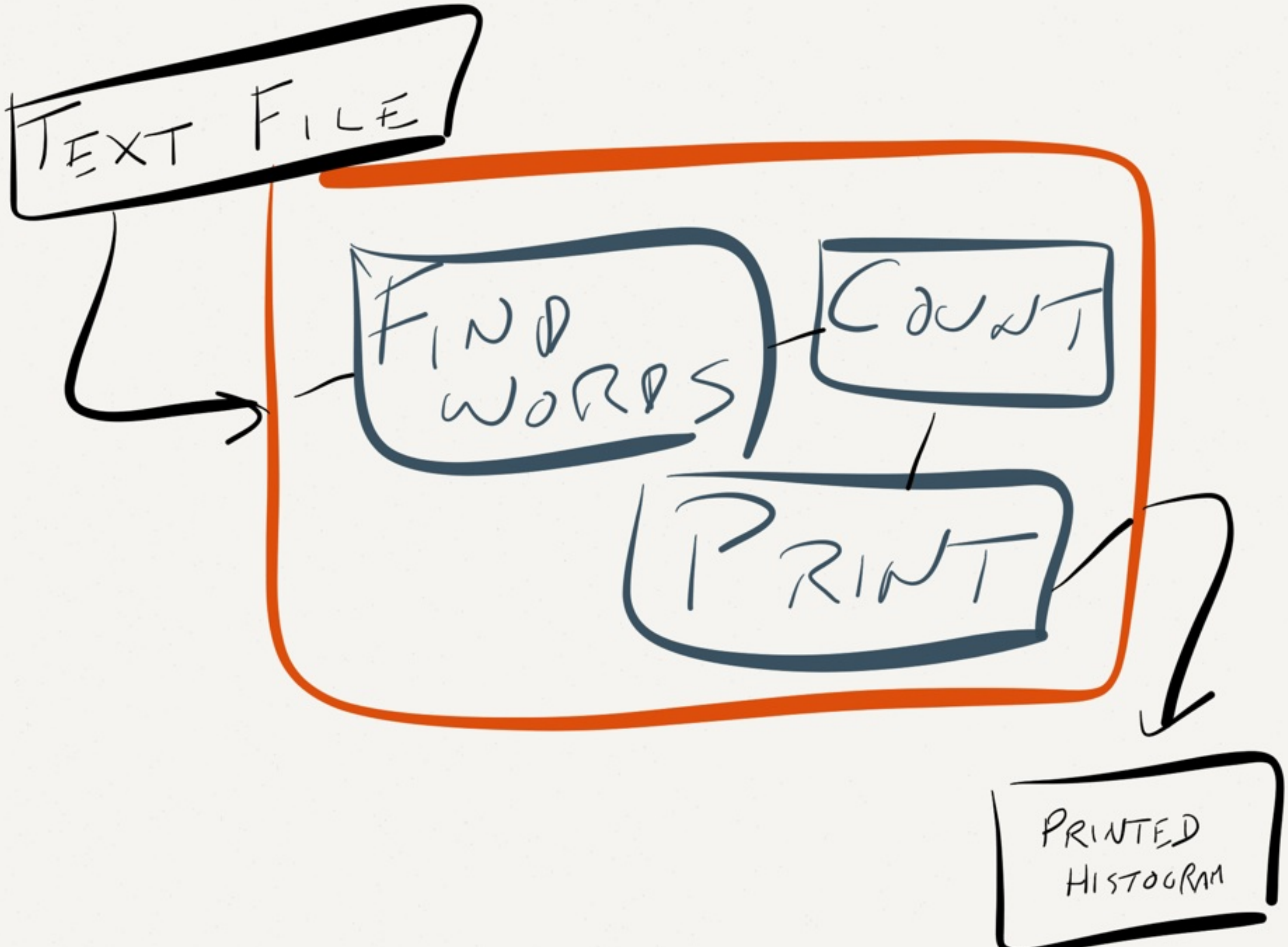
Histogram result:

The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2

Decompose the problem

- Sub-problems:
  1. How do we iterate through the text file, identifying all of the words?
  2. Once we can produce a stream of words, how do we calculate their frequency?
  3. Once we have calculated the frequencies, how do we print out the result?

- What is the interface between these components?
- Can we test them individually?


TEXT FILE
FIND WORDS
COUNT
PRINT
PRINTED HISTOGRAM

How to produce a stream of words?

1. How do we iterate through the text file, identifying all of the words?

```
public interface Iterator<T> {
    // returns true if the iteration has more elements
    public boolean hasNext();

    // returns the next element in the iteration
    public T next();
}
```

- **Key idea:** Define a class (WordScanner) that implements this interface by reading words from a text file.

Coding: Histogram.java

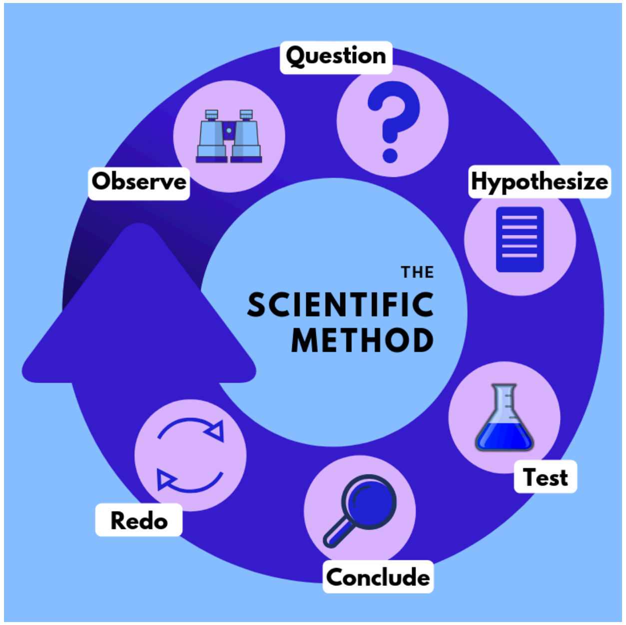
WordScanner.java

Histogram.java

Some Advice on Debugging

Use the Scientific Method

1. Make an observation / ask a question
   - One of my test cases fails!
   - Which assertion? What exception? What is the stack trace?
2. Formulate a hypothesis
   - Could I have passed null as bar to foo.munge(bar)?
3. Conduct an experiment
   - Modify the program to try to confirm or refute the hypothesis.
   - *Don't* make random changes!
   - Predict the outcome of your experiment
   - Re-run test cases, or execute the program
4. Analyze the results
   - Did the modified code behave as expected?
5. Draw conclusions / Report results
   - Create a new test case (if appropriate)



Observing Behavior

- Understand exceptions and their stack traces
  - They give you a lot of information

- If you are using Eclipse, it is worth taking a little time to learn how to use the debugger!
  - See Piazza for a Quick Start tutorial

- Simple print statements are also very effective!
  - Confirm or disprove hypothesis
  - e.g.: The code reached "HERE!" (or not)