

CMPE 152: Compiler Design

November 30 Class Meeting

Department of Computer Engineering
San Jose State University



Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak



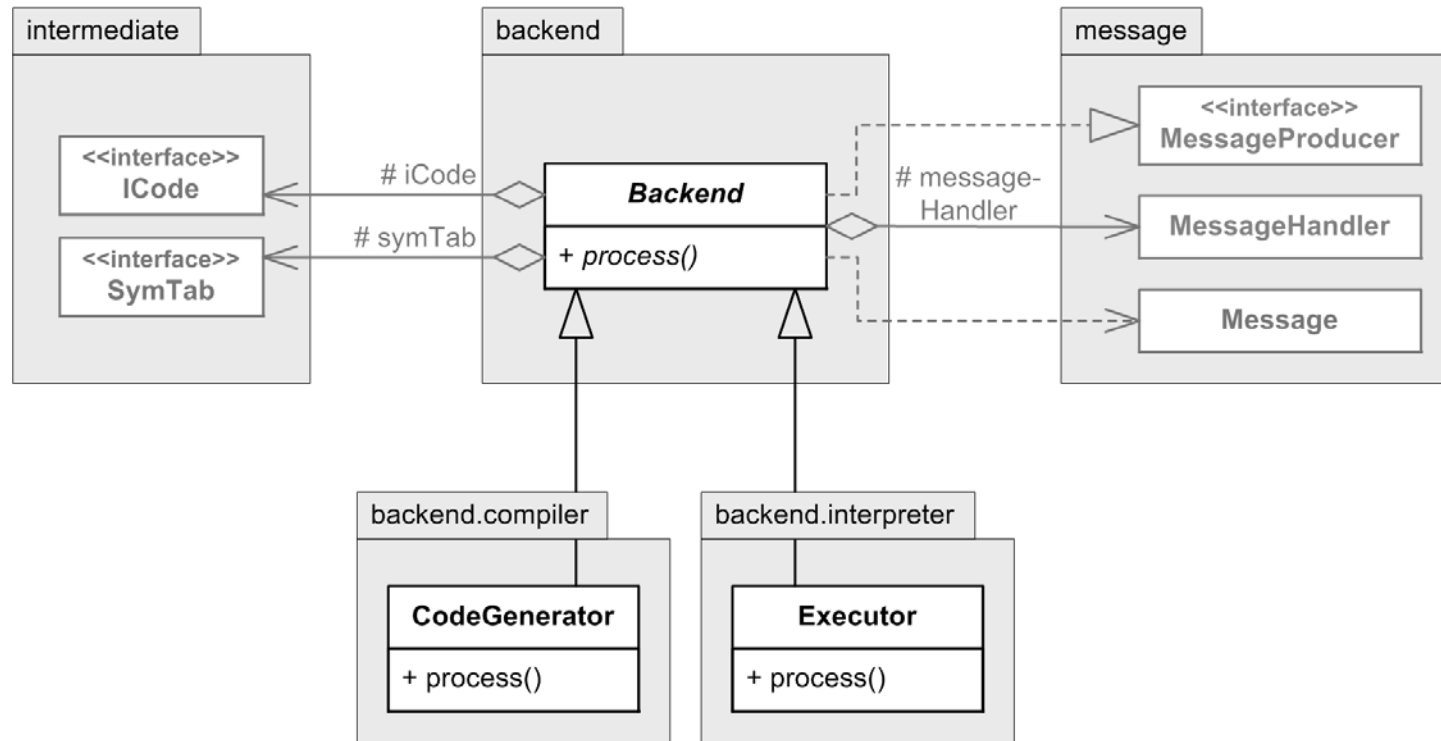
Extra-Credit Oral Presentations

- ❑ Let me know if your team would like to do an oral presentation for extra credit.
- ❑ Tell about your language.
 - Show some example programs.
- ❑ Describe its grammar
 - Show syntax diagrams.
- ❑ What Jasmin code do you generate?
 - Show some code diagrams.
- ❑ Demo: Compile, execute, and run some sample programs.

Extra-Credit Oral Presentations, *cont'd*

- ❑ Submit a note by Friday, Dec. 1 into Canvas: [Assignments | Miscellaneous | Presentation](#) if your team wants to present.
 - Choose either Dec. 5 or 7 to present.
- ❑ 15-20 minutes.
- ❑ Can add up to 50 points to each team member's total assignment score.

An Interactive Source-Level Debugger



- ❑ Control the **interpreter** at run time.
- ❑ Straightforward to implement within our framework.

Source-Level Debugger, *cont'd*

- When you're an interpreter, you're in complete control at run time of the source program's execution.
 - You can start and stop the execution.
 - You can examine and modify values of variables.
 - You have access to the entire runtime stack.

Machine-Level vs. Source-Level Debugging

□ Machine level

- Low level, close to the machine language.
- Single stepping: Execute one machine (or assembly) instruction at a time.
- Monitor and set the values of machine registers.

Machine-Level vs. Source-Level Debugging

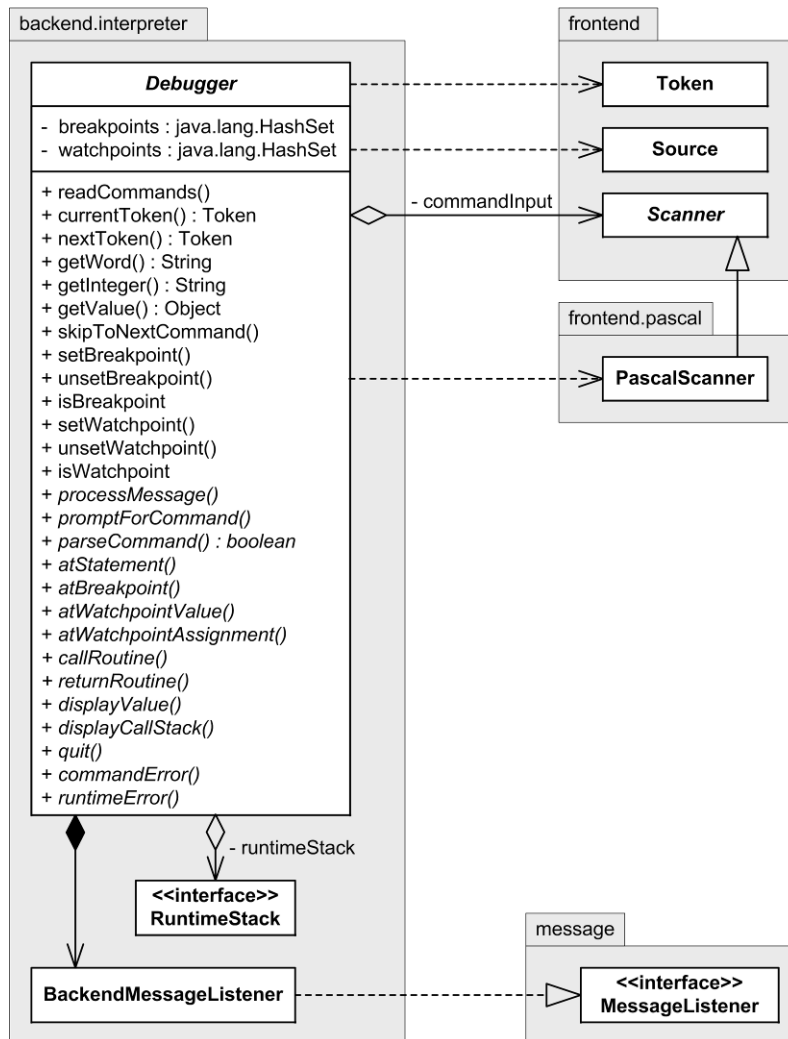
□ Source level

- Debug at the high level of the source language.
- Refer to variables by their names in the source program.
- Refer to statements by their source line numbers.
- Refer to procedures and functions by their names.
- Access to the runtime stack.
- AKA: **symbolic debugger**

Simple Debugger Command-Line Language

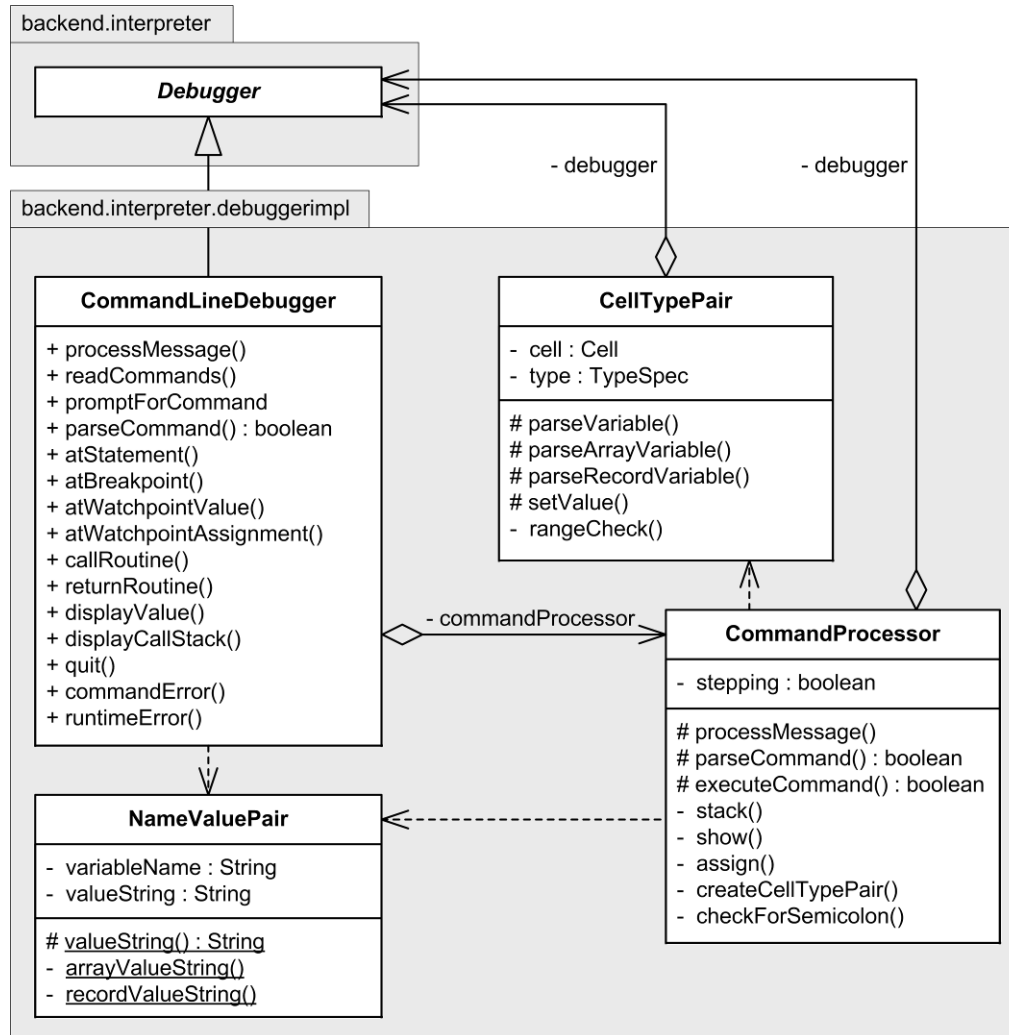
- ❑ Breakpoints: Pause program execution at certain statements.
- ❑ Watchpoints: Monitor the values of certain variables.
- ❑ Assignments: Change the values of variables.
- ❑ Single-step source program execution statement by statement.
- ❑ Display or set the value of a variable.
- ❑ Display the call stack with local values of each routine.

Debugger Architecture



- **Debugger** is an abstract class with two subclasses:
 - **CommandLineDebugger**
 - **GUIDebugger**
- The **Debugger** class listens to messages from the back end.
 - Formerly, the main **Pascal** class listened to the back end messages.
- We need a parser for the command language.
 - Reuse the scanner and token classes from the front end!

Command Line Debugger Architecture



- **CommandProcessor**
 - Processes messages from the back end.
 - Parses debugger commands.
- **CellTypePair**
 - Keeps track of each memory cell and the data type of its value.
 - Parses variables in debugger commands.
- **NameValuePair**
 - Displays the current value of a variable given its name.

Back End Messages

- Messages sent by the interpreter during run time:

Message type	Sent by the back end ...
<code>SOURCE_LINE</code>	Just before executing each statement.
<code>FETCH</code>	Whenever any variable's value is accessed.
<code>ASSIGN</code>	Whenever any variable's value is set.
<code>CALL</code>	Whenever a procedure or function is called.
<code>RETURN</code>	Upon returning from a procedure or function.
<code>RUNTIME_ERROR</code>	Whenever a runtime error occurs.

Method StatementExecutor.execute()

```
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    // Send a message about the current source line.
    sendSourceLineMessage(node);

    switch (nodeType) {

        case COMPOUND: {
            CompoundExecutor compoundExecutor = new CompoundExecutor(this);
            return compoundExecutor.execute(node);
        }
        ...
    }
}
```

- Send a **SOURCE_LINE** message before executing each statement.
- The listener for this message is the debugger.

Method `CommandProcessor.processMessage()`

```
protected void processMessage(Message message)
{
    MessageType type = message.getType();

    switch (type) {

        case SOURCE_LINE: {
            int lineNumber = (Integer) message.getBody();

            if (stepping) {
                debugger.atStatement(lineNumber);
                debugger.readCommands();
            }
            else if (debugger.isBreakpoint(lineNumber)) {
                debugger.atBreakpoint(lineNumber);
                debugger.readCommands();
            }

            break;
        }

        ...

    }
}
```

Single-stepping.

Hit a breakpoint.

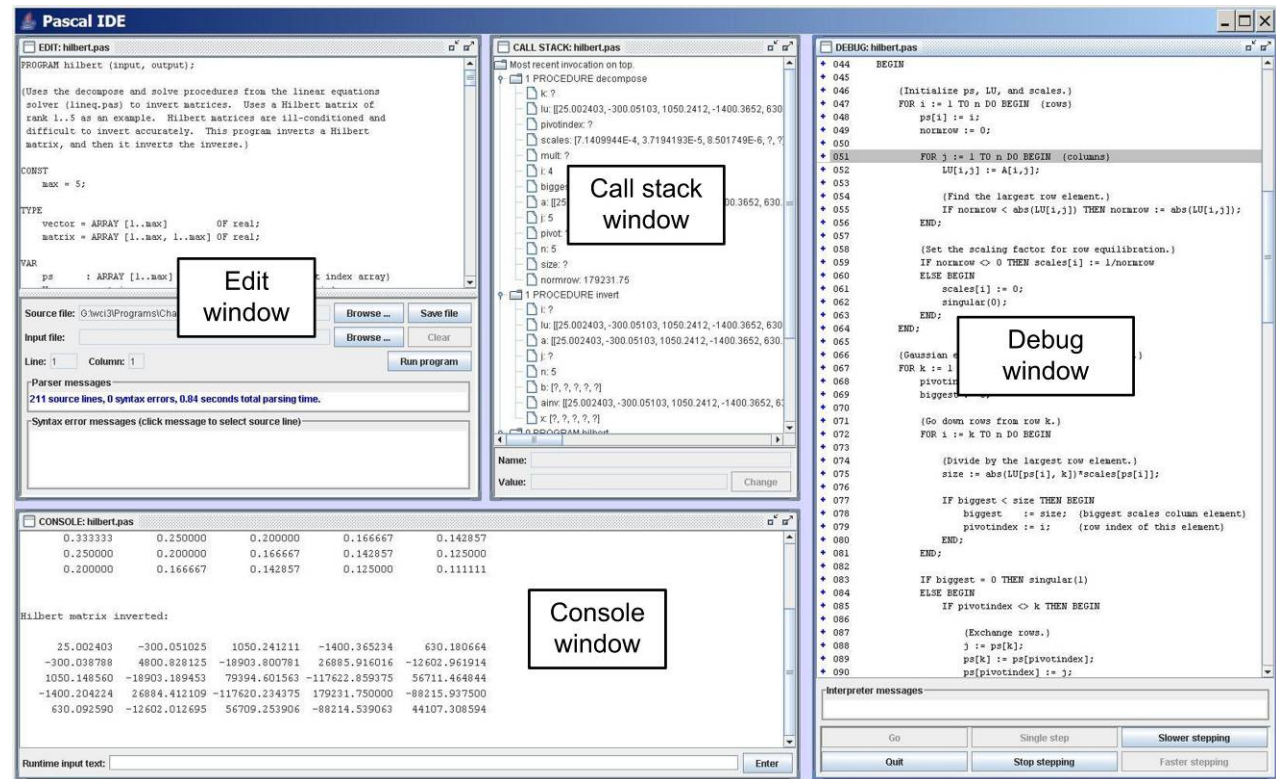
Demo

A Really Cool Debugger ...

- ❑ ... would have a **graphical user interface** (GUI).
- ❑ It would be part of a complete **integrated development environment** (IDE).
 - Different windows for editing, executing, monitoring, input and output, etc.
 - Buttons to invoke debugger operations.
 - Animate the execution of a program.
- ❑ Just like Eclipse!
 - Only perhaps not quite as good.

Integrated Development Environment (IDE)

- A graphical user interface (GUI) that integrates:
 - Edit window
 - Debug window
 - Call stack window
 - Console window
- Implemented with the Java Foundation Classes (Swing)
- Uses multithreaded programming.

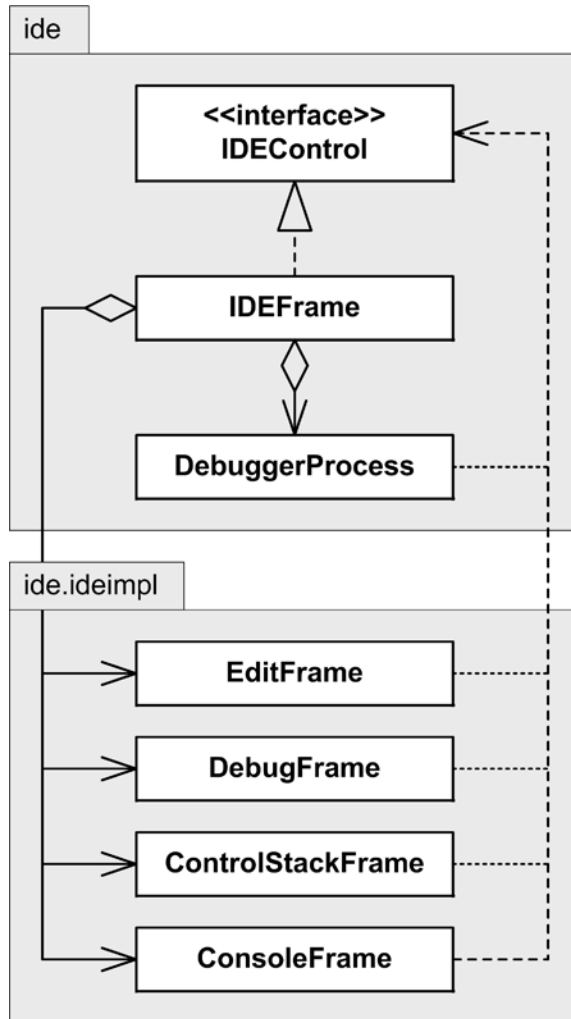


“Wrap a GUI” around the command-line debugger.

The Basic Idea Behind the IDE

- ❑ Run the Pascal **command-line debugger** in one process.
- ❑ Run the IDE **GUI code** in another process.
- ❑ The user manually performs an action on the IDE GUI (e.g., click the Step button)
 - The IDE sends the appropriate **command-line command** to the debugger.
 - The **debugger** reads the command from its **standard input** just as if the user had typed it on the command line.

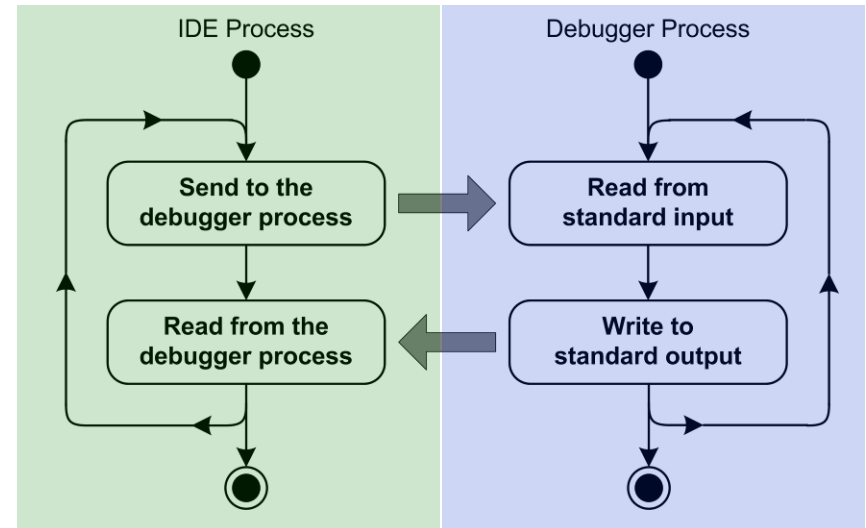
The IDE Framework



- ❑ Multithreaded programming:
 - The IDE GUI code runs in the **IDE process**.
 - The debugger code runs in the **debugger process**.

Interprocess Communication

- The **IDE process** sends debugger commands to the **debugger process**.
 - The **IDE** writes the commands to its **standard output**.
 - The **debugger** reads the commands via its **standard input**.
- The **debugger process** sends status information or program output to the **IDE process**.
 - The **debugger** writes to its **standard output**.
 - The **IDE** reads the debugger's output via its **standard input**.



- The debugger running in the **debugger process** believes that it's reading debugger commands typed on the command line and that it's writing to the console.

Recall: Class CommandLineDebugger

```
public class CommandLineDebugger extends Debugger
{
    ...

    public void atStatement(Integer lineNumber)
    {
        System.out.println("\n>>> At line " + lineNumber);
    }

    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println("\n>>> Breakpoint at line " + lineNumber);
    }

    ...
}
```

- The command-line debugger writes messages to the console (via its standard output) for the user.

Compare to: Class GUIDebugger

```
public class GUIDebugger extends Debugger
{
    ...

    public void atStatement(Integer lineNumber)
    {
        System.out.println(DEBUGGER_AT_TAG + lineNumber);
    }

    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println(DEBUGGER_BREAK_TAG + lineNumber);
    }

    ...
}
```

□ The GUI debugger writes **tagged messages** to the “console” (via its standard output) for the GUI process.

Tagged Messages for the GUI Process

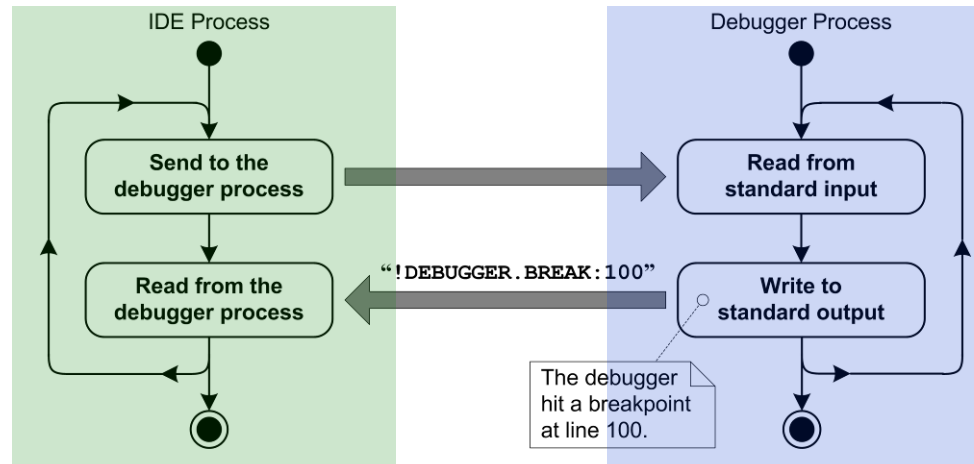
```
package wci.ide;

public interface IDEControl
{
    // Debugger output line tags.
    public static final String LISTING_TAG = "!LISTING: ";
    public static final String PARSER_TAG = "!PARSER: ";
    public static final String SYNTAX_TAG = "!SYNTAX: ";
    public static final String INTERPRETER_TAG = "!INTERPRETER: ";

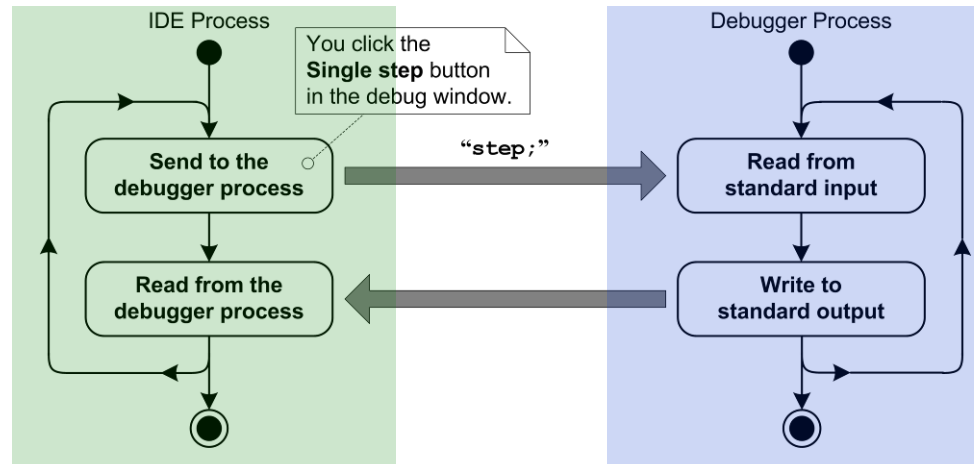
    public static final String DEBUGGER_AT_TAG = "!DEBUGGER.AT: ";
    public static final String DEBUGGER_BREAK_TAG = "!DEBUGGER.BREAK: ";
    public static final String DEBUGGER_ROUTINE_TAG = "!DEBUGGER.ROUTINE: ";
    public static final String DEBUGGER_VARIABLE_TAG = "!DEBUGGER.VARIABLE: ";

    ...
}
```

Interprocess Communication, *cont'd*



Interprocess Communication, *cont'd*



Interprocess Communication, *cont'd*

