

7 The Integrated Debugger

Your skill set for writing programs would not be complete without knowing how to use a debugger. While a debugger is traditionally associated with finding bugs, it can also be used as a general aid for understanding your program as you develop it. jGRASP provides a highly visual debugger for Java, which is tightly integrated with the CSD and UML windows, the Workbench, and the Viewers. The jGRASP debugger includes all of the traditional features expected in a debugger.

If the example program used in this section is not available to you, or if you do not understand it, simply substitute your own program in the discussion.

Objectives – When you have completed this tutorial, you should be able to set breakpoints and step through the program, either by single stepping or auto stepping. You should also be able to display the dynamic state of objects created by the program using the appropriate Object Viewer.

The details of these objectives are captured in the hyperlinked topics listed below.

7.1 Preparing to Run the Debugger

7.2 Setting a Breakpoint

7.3 Running a Program in Debug Mode

7.4 Stepping Through a Program – the Debug Buttons

7.5 Stepping Through a Program – without *Stepping In*

7.6 Stepping Through a Program – and *Stepping In*

7.7 Opening Object Viewers

7.8 Debugging a Program

7.1 Preparing to Run the Debugger

In preparation to use the debugger, we need to make sure that programs are being compiled in debug mode. This is the default, so this option is probably already turned on. With a CSD or UML window in focus, click **Build** on the menu and make sure **Debug Mode** is checked. If the box in front of Debug Mode is not checked, click on the box. When you click on Build again, you should see that Debug Mode is checked. When you compile your program in Debug Mode, information about the program is included in the .class file that would normally be omitted. This allows the debugger to display useful details as you execute the program. If your program has not been compiled with Debug Mode checked, you should recompile it before proceeding.

7.2 Setting a Breakpoint

In order to examine the state of your program at a particular statement, you need to set a breakpoint. The statement you select must be “executable” rather than a simple declaration. To set a breakpoint in a program, move the mouse to the line of code and left-click the mouse to move the cursor there. Now right-click on the line to display a set of options that includes **Toggle Breakpoint**. For example, in Figure 7-1 the cursor is on

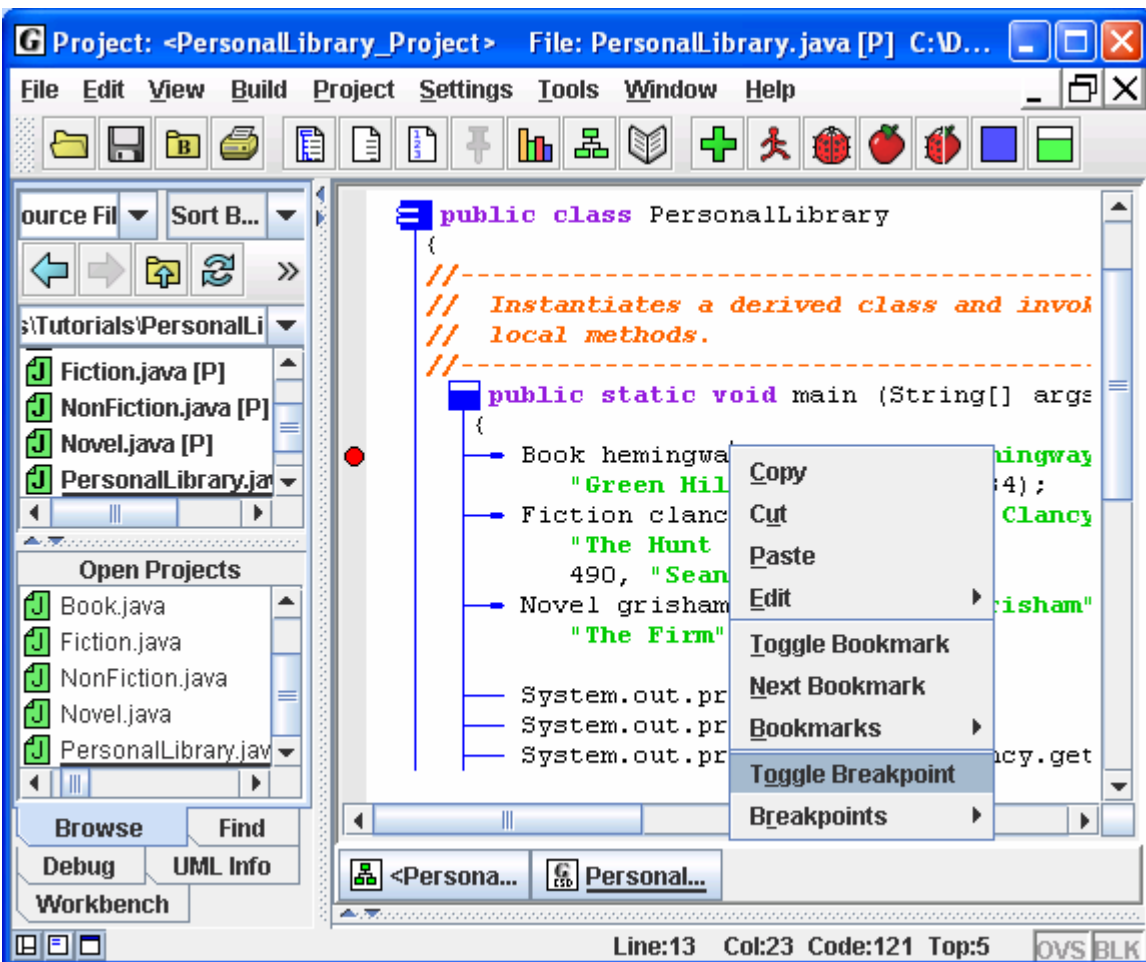





Figure 7-1. Setting a breakpoint

the first executable line in *main* (which declares `Book hemingway ...`), and after **Toggle Breakpoint** is selected in the options popup menu, a small red stop sign symbol  appears in the left margin of the line to indicate that a breakpoint has been set. To remove a breakpoint, you repeat the process since this is a toggle action. You may set as many breakpoints as needed.


You can also set a breakpoint by hovering the mouse over the leftmost column of the line where you want to set the breakpoint. When you see the red octagonal breakpoint symbol , you just left-click the mouse to set the breakpoint. You can remove a breakpoint by clicking on the red octagonal. This second approach is the one most commonly used for setting and removing breakpoints.

7.3 Running a Program in Debug Mode

After compiling your program in Debug Mode and setting one or more breakpoints, you are ready to run your program with the debugger. You can start the debugger in one of two ways:

- (1) Click **Build – Debug** on the CSD window menu, or
- (2) Click the Debug button  on the toolbar.

After you start the debug session, several things happen. In the Run window near the bottom of the Desktop, you should see a message indicating the debugger has been launched. In the CSD window, the line with the breakpoint set is eventually highlighted, indicating that the program will execute this statement next. On the left side of the jGRASP desktop, the Debug tab is popped to the top. Each of these can be seen in Figure 7-2. Notice the Debug tab pane is further divided into three sub-panes or sections labeled **Threads**, **Call Stack**, and **Variables/Eval**. Each of these sections can be resized by selecting and dragging one of the horizontal partitions.

The **Threads** section lists all of the active threads running in the program. In the example, the red thread symbol  indicates the program is stopped in *main*, and green indicates a thread is running. Advanced users should find this feature quite useful for starting and stopping individual threads in their programs. However, since beginners and intermediate users rarely use multi-threading, the thread section is closed when the debugger is initially started. Once the Threads section is dragged open, it remains open for the duration of the jGRASP session.

The **Call Stack** section is useful to all levels of users since it shows the current call stack and allows the user to switch from one level to another in the call stack. When this occurs, the CSD window that contains the source code associated with a particular call is popped to the top of the desktop.

The **Variables/Eval** section shows the details of the current state of the program in the *Variables* tab and provides an easy way to evaluate expressions involving these variables in the *Eval* tab. Most of your attention will be focused on the *Variables* tab where you can monitor all current values in the program. From the *Variables* tab, you can also launch separate viewers on any primitives or objects as well as fields of objects.

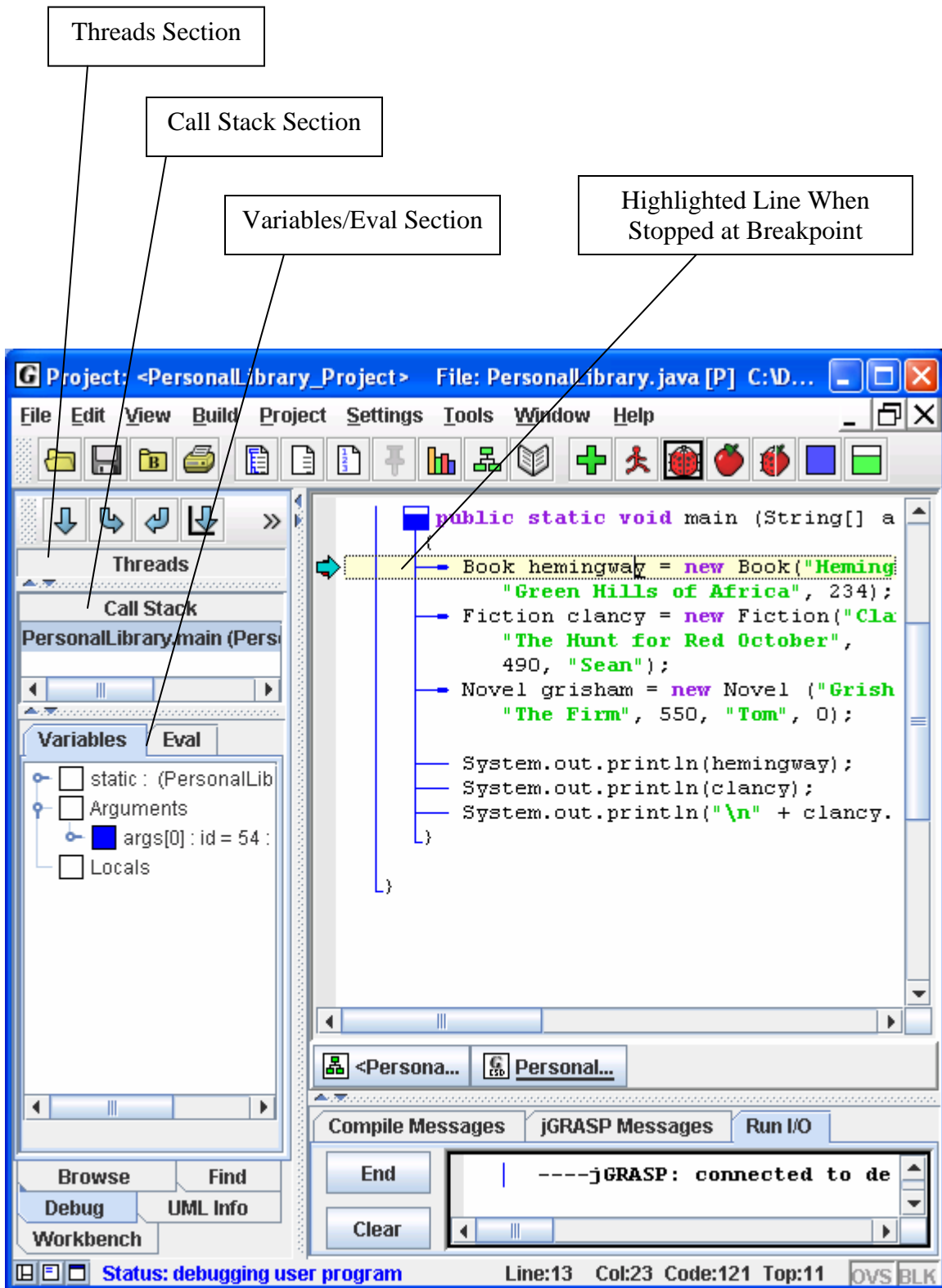














Figure 7-2. Desktop after debugger is started

7.4 Stepping Through a Program – the Debug Buttons




After the program stops at the breakpoint (Figure 7-2), you can use the buttons at the top of the Debug tab to *step*, *step into* a method call, *step out* of a method, *run to the cursor*, *pause* the current thread, *resume*, turn on/off *auto step* mode, turn on/off *auto resume* mode, and *suspend* new thread. Note that an application program begins at the first executable statement in the *main* method. The sequence of statements that is executed when you run your program is called the *control path* (or simply *path*). If your program includes statements with conditions (e.g., *if* or *while* statements), the control path will reflect the *true* or *false* state of the conditions in these statements.

- ↓ Clicking the *Step* button will single step to the next statement. The highlighted line in the CSD window indicates the statement that's about to be executed. When the Step button is clicked, that statement is executed and the "highlighting" is moved to the next statement along the control path.
- ↪ Clicking the *Step in* button for a statement with a method call that's part of the user's source code will open the new file, if it's not already open, and pop its CSD window to the top with the current statement highlighted. The top entry in the *Call Stack* indicates where you are in the program. Note that clicking the *Step in* button for a statement without a method call is equivalent to clicking *Step*.
- ↪ Clicking the *Step out* button will take to the statement in the CSD window from which you previously stepped in. The *Call Stack* will be updated accordingly.
- ↪ Clicking the *Run to Cursor* button will cause your program to step automatically until the statement with the cursor L is reached. If the cursor is not on a statement along the control path, the program will stop at the next breakpoint it encounters or at the end of the program. The *Run to Cursor* button is convenient since placing the cursor on a statement is like setting "temporary" breakpoint.
- ⏏ Clicking the *Pause* button suspend the program running in debug mode. Note that if you didn't have a breakpoint set in your code, you may have to select the main thread in the *Threads* section before the *Pause* button is available. After the program has halted, refer to the *Call Stack* and select the last method in your source code that was invoked. This should open the CSD window containing the method with the current line highlighted. Place a breakpoint on the next line and click the *step* ↓ button to advance through the code.
- Clicking the *Resume* button advances the program along the control path to the next breakpoint or to the end of the program. If you have set a breakpoint in a CSD window containing another file and this breakpoint is on the control path (i.e., in a method that gets called), then this CSD window will pop to the top when the breakpoint is reached.

-  The *Auto Step* button is used to toggle off and on a mode which allows you to step repeatedly after clicking the *step*  button only once. This is an extremely useful feature in that it essentially let's you watch your program run. Notice that with this feature turned on, a *Delay* slider bar appears beneath the Debug controls. This allows you to set the delay between steps from 0 to 26 seconds (default is .5 seconds). While the program is auto stepping, you can stop the program by clicking the *Pause*  button. Clicking the *Step*  button again continues the auto stepping. Remember after turning on Auto Step , you always have to click the *step*  button once to get things rolling.
-  The *Auto Resume* button is used to toggle off and on a mode which allows you to resume repeatedly after clicking the *Resume*  button only once. The effect is that your program moves from breakpoint to breakpoint using the delay indicated on the *delay* slider bar. As with auto step above, you can click the *Pause*  button to interrupt the auto resume; then clicking the *Resume*  button again continues the auto resume.
-  The *Use Byte Code Size Steps* button toggles on and off the mode that allows you to step through a program in the smallest increments possible. With this feature off, the step size is approximately one source code statement, which is what most users want to see. This feature is seldom needed by beginning and intermediate programmers.
-  The *Suspend New Threads* button toggles on and off the mode that will immediately suspend any new threads that start. With this feature on when the debugging process is started, all startup threads are suspended as soon as is possible. Unless you are writing programs with multiple threads, you should leave the feature turned off.

As you move through the program, you can watch the call stack and contents of variables change dynamically with each step. The integrated debugger is especially useful for watching the creation of objects as the user steps through various levels of constructors. The jGRASP debugger can be used very effectively to explain programs, since a major part of understanding a program is keeping track (mentally or otherwise) of the state of the program as one reads from line to line. We will make two passes through the example program as we explain it. During the first pass, we will “step” through the program without “stepping into” any of the method calls, and we will concentrate on the Variable section.

7.5 Stepping Through a Program – without *Stepping In*

After initially arriving at the breakpoint in Figure 7-2, the **Variables/ Settings** section indicates no local variables have been declared. Figure 7-3 shows the results of clicking the *Step*  button to move to the next statement. Notice that under Locals in the **Variables/Eval** section, we now have an instance of *Book* called *hemingway*. Objects, represented by a colored square, can be opened and closed by clicking the “handle” in front of the square object. Primitives, like the integer *pages*, are represented by colored triangles. In Figure 7-3, *hemingway* has been opened to show the author, title, and pages fields. Each of the String instances (e.g., author) can be opened to show the details of a String object, including the character array that holds the actual value of the string.

Since *hemingway* is an instance of *Book*, the fields in *hemingway* are marked with green object or primitive symbols to indicate that they were declared in *Book*. Notice that the symbols for *author* and *title* have red borders since they were declared to be *private* in *Book* to indicate they are inaccessible from the current context of *main* in *PersonalLibrary*. The field *pages*, which was declared to be *protected* in *Book*, has a symbol without a red border. The reason for this is somewhat subtle. The *protected* field *pages* is accessible in all subclasses of *Book* as well as in any class contained the Java package containing *Book*. Since the *PersonalLibrary* program is not in a package, the directory containing it is considered the “package.” Thus, since *Book* is in the same directory as *PersonalLibrary*, the *protected* field *pages* is accessible to *PersonalLibrary*.

After executing the statement indicated in Figure 7-3, an instance of the *Fiction* class called *clancy* is created as shown in Figure 7-4. In the figure, *clancy* has been opened to reveal its fields. The field “*mainCharacter*” is green, indicating it is defined in *Fiction*.

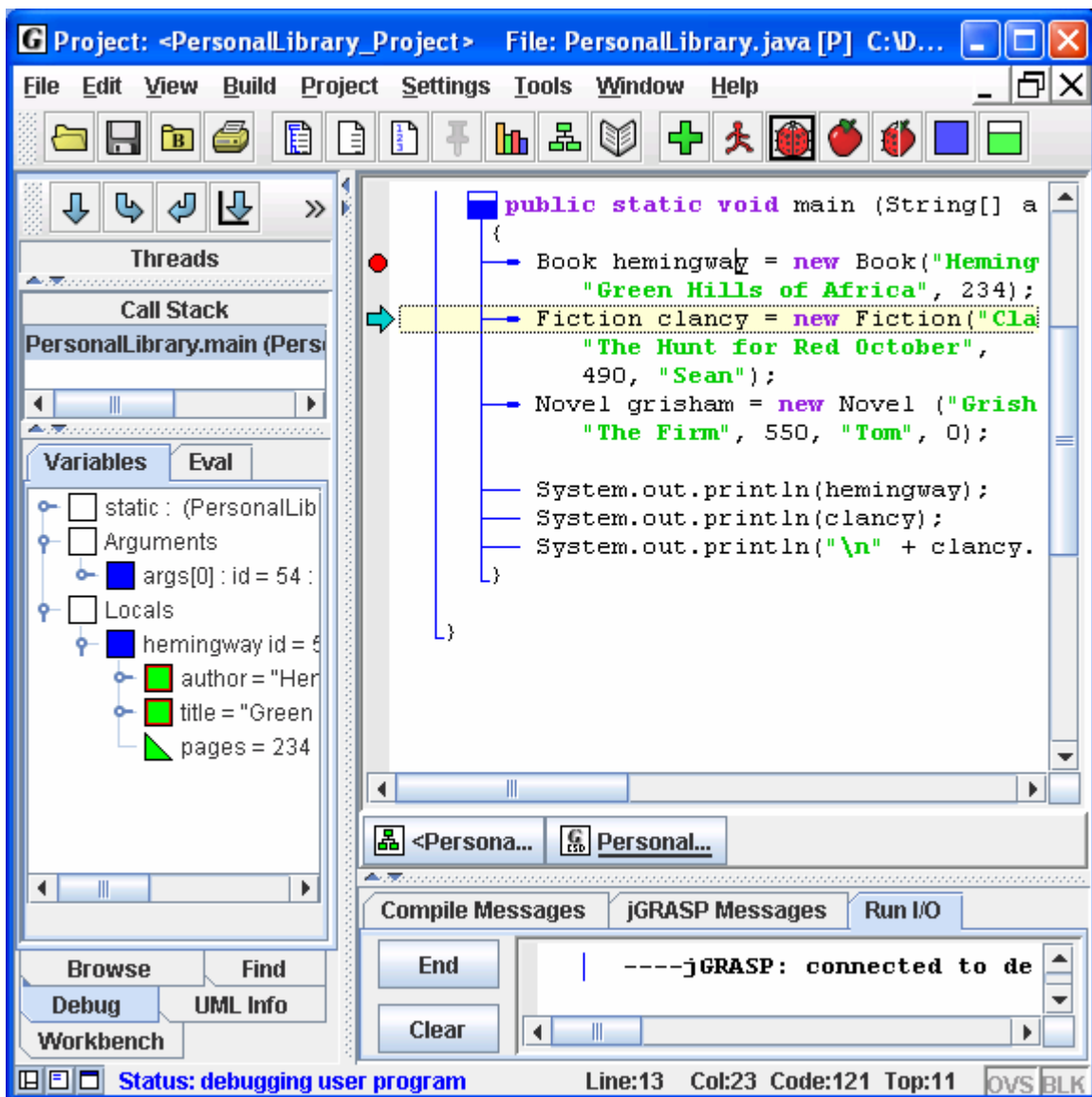


Figure 7-3. Desktop after hemingway (book) is created

The other fields (author, title, and pages) are orange, which indicates these fields were inherited from Book.

As you continue to step through your program, you should see output of the program displayed in the Run I/O window in the lower half of the Desktop. Eventually, you should reach the end of the program and see it terminate. When this occurs, the Debug tab should become blank, indicating that the program is no longer running.

7.6 Stepping Through a Program – and *Stepping In*

Now we are ready to make a second pass and “step in” to the methods called. Tracing through a program by following the calls to methods can be quite instructive in the obvious way. In the object-oriented paradigm, it is quite useful for illustrating the concept of constructors. As before, we need to run the example program in the debugger

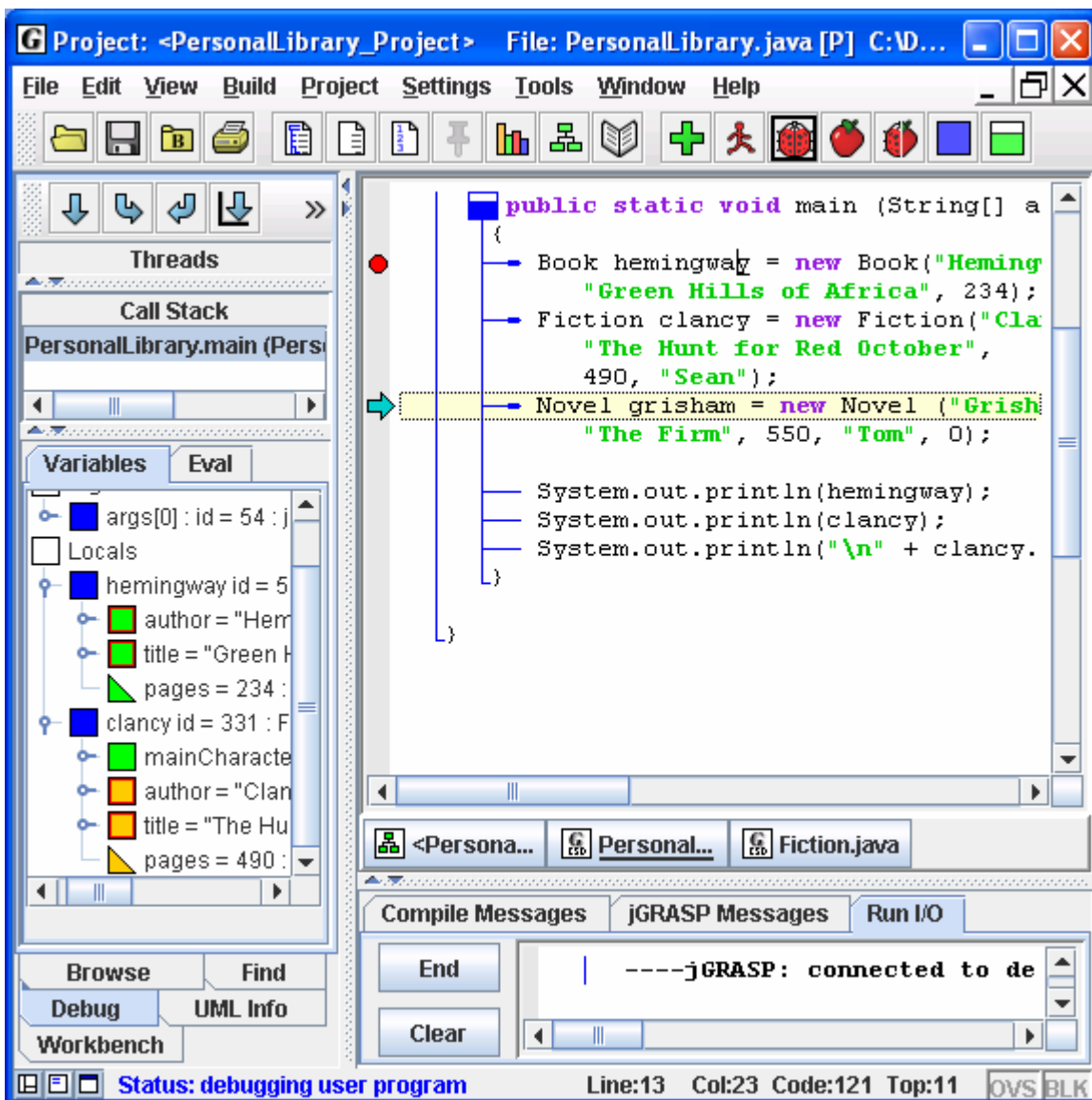




Figure 7-4. After next step and "clancy" created

by clicking **Build – Debug** on the CSD window menu or by clicking the debug button  on the toolbar. After arriving at the breakpoint, we click the *Step in* button  and the constructor for class `Book` pops up in the CSD window (Figure 7-5). Notice the *Call Stack* in the Debug tab indicates you have moved into `Book` from `PersonalLibrary` (i.e., the entry for `Book` is listed above `PersonalLibrary` in the *call stack*). If you click on the `PersonalLibrary` entry in the call stack, the associated CSD window will pop to the top and you see the variables associated with it. If you then click the `Book` entry, its CSD window pops to the top and you see the variables associated with the call to `Book`'s constructor. In Figure 7-5, the entry for *this* has been expanded in the *Variables* section. The *this* object represents the object that is being constructed. Notice none of the fields have a red border since we are inside the `Book` class. As you step through the constructor, you should see the fields in this get initialized to the values passed in as arguments. Also, note the *id* for *this* (it is 325 in our example debug session; it may be a

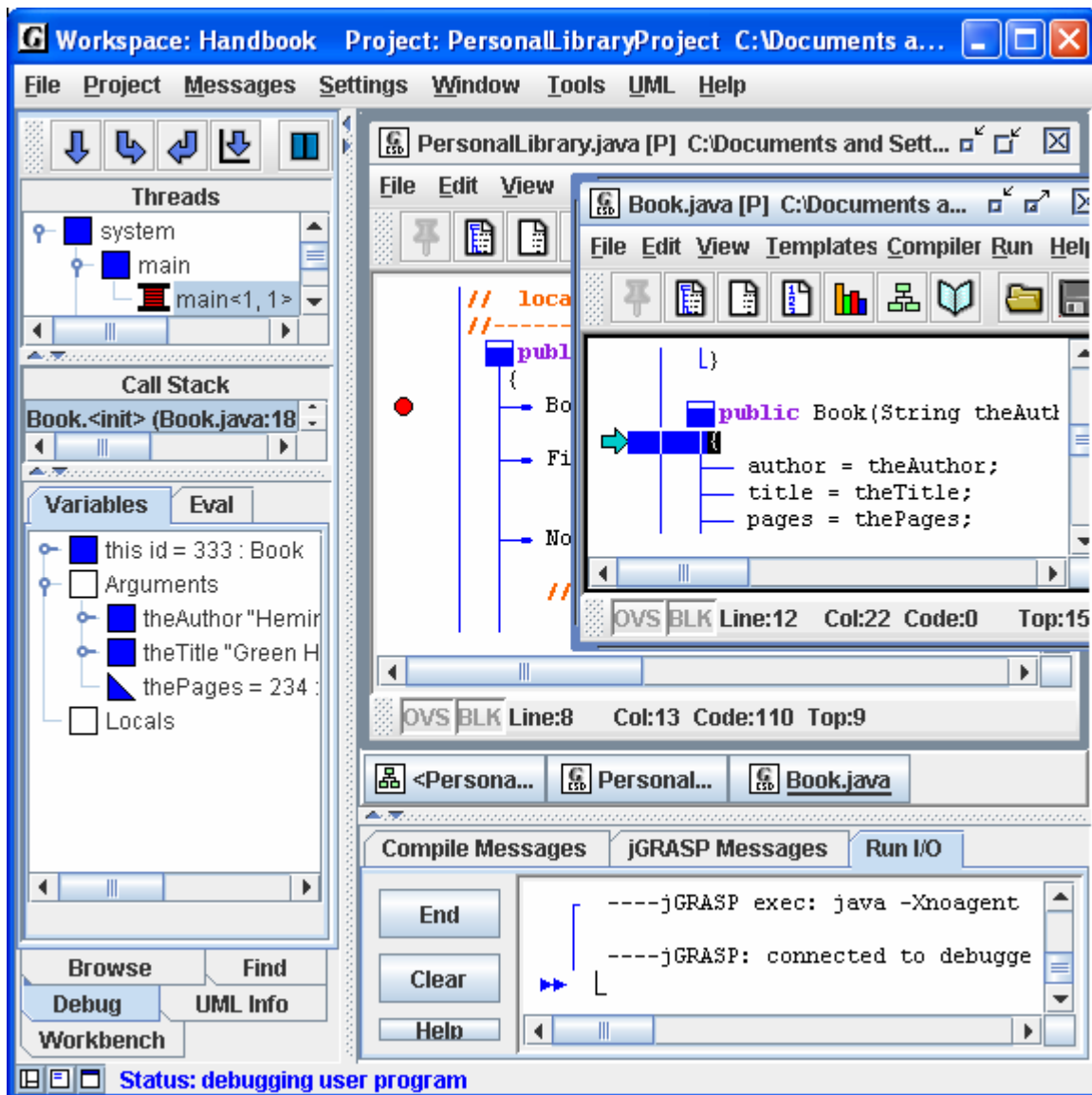


Figure 7-5. After next stepping into the `Book` constructor

different number in your session). You can then step through the constructor in the usual way, eventually returning to the statement in the main program that called the constructor. One more step should finally get you to the next statement, and you should see *hemingway* in the *Variables* section with the same *id* as you saw in the constructor as it was being built. If you expand *hemingway*, you should see the red borders are back on *author* and *title* since we're no longer in book class.

There are many other scenarios where this approach of tracing through the process of object construction is useful and instructive. For example, consider the case where the Fiction constructor for "clancy" is called and it in turn calls the super constructor located in Book. By stepping into each call, you can see not only how the program proceeds through the constructor's code, but also how fields are initialized.

Another even more common example is when the *toString* method of an object is invoked indirectly in a print statement (`System.out.println`). The debugger actually takes the user to the object's respective *toString* method.

7.7 Opening Object Viewers


A separate *Viewer* window can be opened for any primitive or object (or field of an object) displayed in *Variables* section of the Debug tab. All objects have a *basic* view which is the view shown in the Debug tab. However, when a separate viewer window is opened for an entry, some objects will have additional views.

The easiest way to open a viewer is to left-click on an object and drag it from the workbench to the location where you want the viewer to open. This will open a "view by name" viewer. You can also open a viewer by right-clicking on the object and selecting either **View by Value** or **View by Name**.

Figure 7-6 shows an object viewer for the *title* field of *hemingway* in Figure 7-4, which is a String object in an instance of Book. *Formatted* is the default "view" for a String object which is especially useful when viewing a String object with a large value (e.g., a page of text). In Figure 7-7, the *Basic* view has been selected and expanded to show the details of the String object. Notice the first field is `value[21]` which is a character array holding the actual value of the string. If we open a separate viewer on value, we have a *Presentation* view of the array as shown in Figure 7-8. Notice that the first element ('G') in the array has been selected and this opened a subview of type character. The subview displays the 'G' and its integral value of 71. If our example had been an array of strings (e.g., a list of words) then selecting an array element would have displayed the formatted view of a String object in the subview. Presentation view is the default for arrays. There is also a view called *Array Elements* which is quite useful for large arrays.



Figure 7-6. Viewing a String Object

You are encouraged to open separate viewers for any of the primitives and objects in the Variables section of the Debug tab. In addition to providing multiple views of the object, each viewer includes an Invoke Method button  for the object being viewed.

In the tutorial 9 *Viewers for Data Structures*, many other examples are presented along with a more detailed description of viewers in general. The "Structure Identifier" viewer is introduced which attempts to automatically recognize and display linked lists, binary trees, and array wrappers (lists, stacks, queues, etc.) when opened on an object during debugging or workbench use.

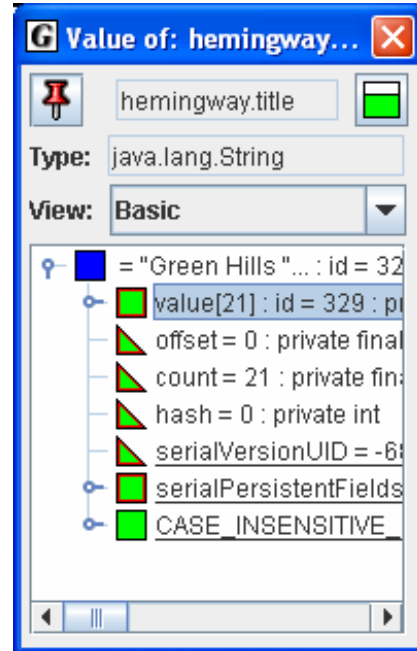


Figure 7-7. Basic view of a string (expanded to see fields)

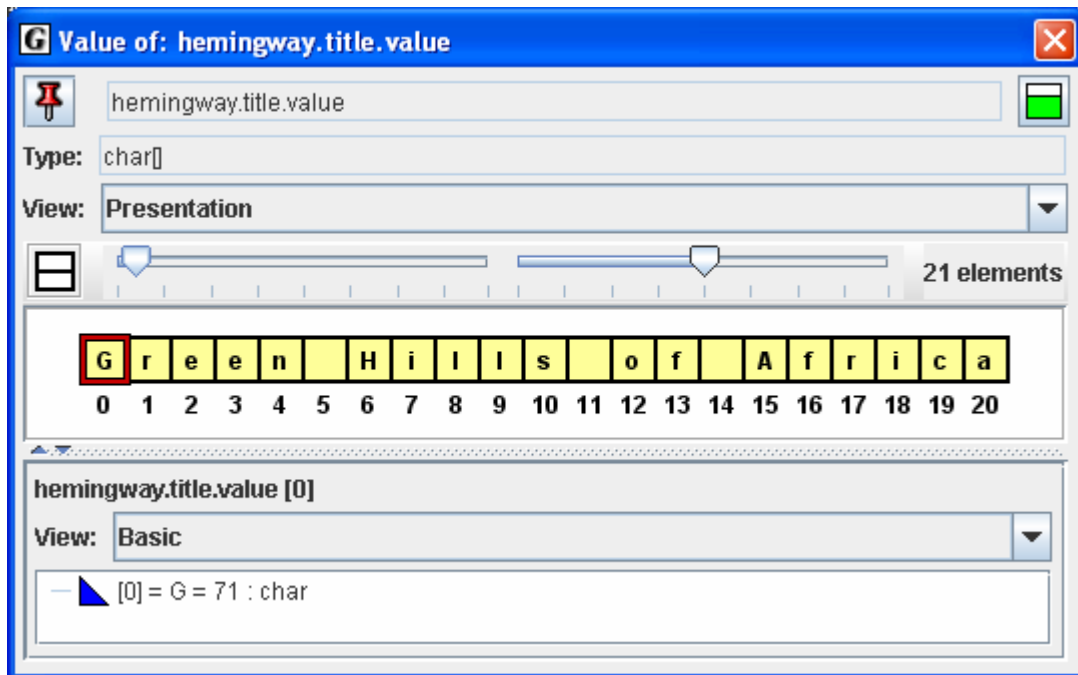



Figure 7-8. Presentation View of hemingway.title.value

7.8 Debugging a Program

You have, no doubt, noticed that the previous discussion was only indirectly related to the activity of finding and removing *bugs* from your program. It was intended to show you how to set and unset breakpoints and how to step through your program. Typically, to find a bug in your program, you need to have an idea where in the program things are going wrong. The strategy is to set a breakpoint on a line of code prior to the line where you think the problem occurs. When the program gets to the breakpoint, you can inspect the variables of interest to ensure that they have the correct values. Assuming the values are okay, you can begin stepping through the program, watching for the error to occur. Of course, if the value of one or more of the variables was wrong at the breakpoint, you will need to set the breakpoint earlier in the program.

You can also set several types of “watches” on a field of an object. In Figure 7-9, a *Watch for Access* has been set on the *title* in *hemingway* just after it was created. If you click the Resume button  at this point, with no breakpoints set before the end of the program, the next place the program should stop is in the *toString* method of *Book* in conjunction with the *println* statement for *hemingway*. This is because the *title* field of *hemingway* is accessed in the statement:

```
return("\nAuthor: " + author +
      "\nTitle: " + title +
      "\nPages: " + pages);
```

Note that setting *Watch All for Access* on the *title* field of *hemingway* sets the *watch* on all occurrences of the *title* field (i.e., in all instances of *Book*, *Fiction*, and *Novel*).

As your programs become more complex, the debugger can be an extremely useful for both understanding your program and isolating bugs. For additional details, see *Integrated Java Debugger* in **jGRASP Help**.

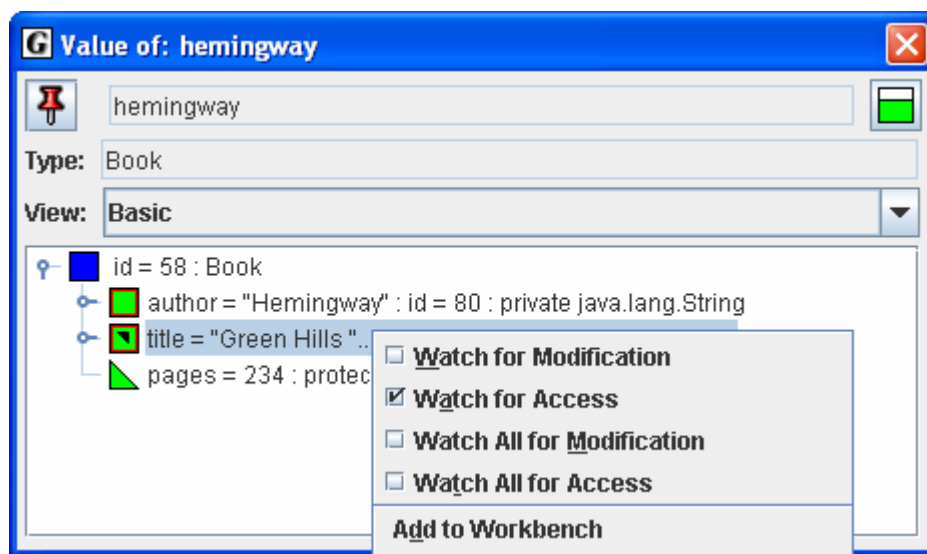


Figure 7-9. Setting a Watch for Access