

---

---

# Harlequin Dylan

## Getting Started with Harlequin Dylan.

Version 1.0



## Copyright and Trademarks

*Harlequin Dylan: Getting Started with Harlequin Dylan*

Version 1.0

March 1998

Part number: DYL-1.0-GE

Copyright © 1997–1998 by Harlequin Group plc.

Companies, names and data used in examples herein are fictitious unless otherwise noted.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

Harlequin Dylan is a trademark of Harlequin Group plc.

Other brand or product names are the registered trademarks or trademarks of their respective holders.

### US Government Use

The Harlequin Dylan Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be 'unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

### Europe:

**Harlequin Limited**  
Barrington Hall  
Barrington  
Cambridge CB2 5RG  
U.K.

telephone +44 1223 873 800  
fax +44 1223 873 873

### North America:

**Harlequin Incorporated**  
One Cambridge Center  
Cambridge, MA 02142  
U.S.A.

telephone +1 617 374 2400  
fax +1 617 252 6505

### Electronic Access:

<http://www.harlequin.co.uk/>  
<http://www.harlequin.com/>

---

---

# Contents

## **1 Quick Start 5**

Starting work with Harlequin Dylan 5  
The project window 7  
The Reversi project 9  
Building an executable application 9  
Three ways of running Dylan applications 13  
Looking at definitions 13  
Making changes 14  
Summary 17

## **2 Fixing Bugs 19**

Rebuilding the application 19  
Problems at compile time 20  
Problems at run time 25  
Summary 36

## **3 Programming in Harlequin Dylan 37**

Projects 37  
Development models 42  
Compilation 43  
Executing programs 46  
Source, database, and run-time views 47  
Summary 48

<b>4</b>	<b>Creating Projects</b>	<b>49</b>
	Creating a new project	49
	New Project wizard persistence	59
	Advanced project settings	60
	Inserting new files and subprojects into a project	60
	Moving files within a project	60
	Deleting files from a project	61
	Project settings	61
	Project files and LID files	62
<b>5</b>	<b>Learning More About an Application</b>	<b>65</b>
	About the browser	65
	Browsing Reversi	66
	Navigation with the browser	70
<b>6</b>	<b>Debugging and Interactive Development</b>	<b>73</b>
	The debugger	73
	About the debugger	73
	Debugging a specific application thread	77
	Controlling application execution	77
	Changing the debugger layout	79
	Setting breakpoints on application code	79
	Saving a bug report	82
	Refreshing debugger views	84
	Debugger options	84
<b>7</b>	<b>Dispatch Optimization Coloring in the Editor</b>	<b>89</b>
	About dispatch optimizations	89
	Optimization coloring	90
	Optimizing the Reversi application	92
	<b>Appendix A The Stand-alone Compiler</b>	<b>97</b>
	<b>Appendix B Editor Options</b>	<b>105</b>

# 1

---

---

## Quick Start

This first chapter is a quick introduction to building applications with Harlequin Dylan.

### 1.1 Starting work with Harlequin Dylan

In this section, we start Harlequin Dylan's integrated development environment, and choose one of several supplied examples to work with.

Start the Harlequin Dylan environment. You can do this by choosing the Start-menu shortcut **Start > Programs > Harlequin Dylan > Dylan**. Harlequin Dylan will also start if you open any file associated with it, such as a Dylan source file.

When Harlequin Dylan starts, the *main window* appears:



Figure 1.1 The Harlequin Dylan main window.

A dialog also appears:

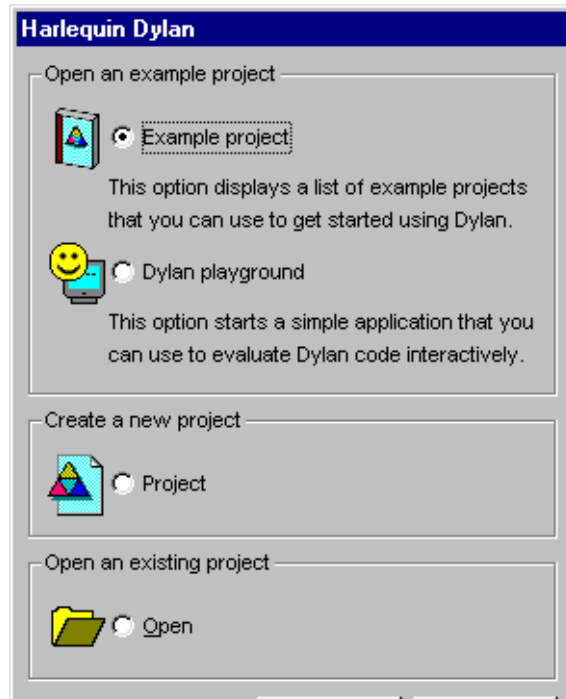


Figure 1.2 The Harlequin Dylan initial dialog.

The Harlequin Dylan main window will be present throughout your Harlequin Dylan session. It provides a means of setting environment-wide options and of controlling the display of all Dylan windows. (Clicking on this window brings all other Dylan windows to the front.)

The dialog that appears is there to help you get working quickly, whether by looking at an example project, creating a new project, opening an existing project or text file, or starting an interactive Dylan session.

We want to browse the examples.

1. Select the “Example project” option and click **OK**.

The Open Example Project dialog appears.

The Open Example Project dialog shows several categories of example Dylan application. If you expand a category you can see the examples offered. We are going to look at the example called Reversi, which you can find under “Documentation”.

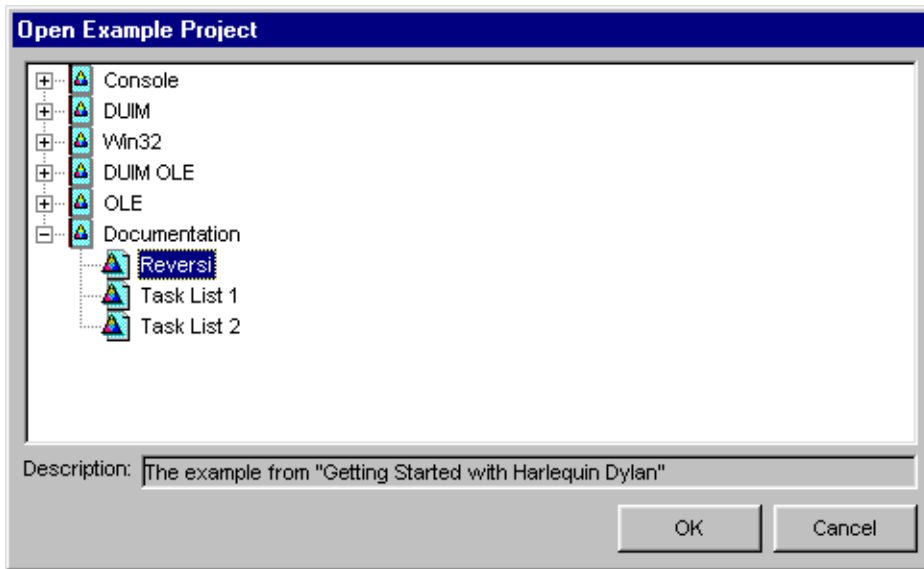


Figure 1.3 The Harlequin Dylan Examples dialog.

2. Select “Documentation“, then “Reversi“, and then click **OK**.

A *project window* appears.

The project window is one of four programming tools in Harlequin Dylan. The other tools in Harlequin Dylan are the browser, the editor, and the debugger. We will see those tools .

## 1.2 The project window

In the project window you can see the project that you are working on. A *project* represents a Dylan library under development. We look at projects in more detail in Chapter 4, “Creating Projects”.

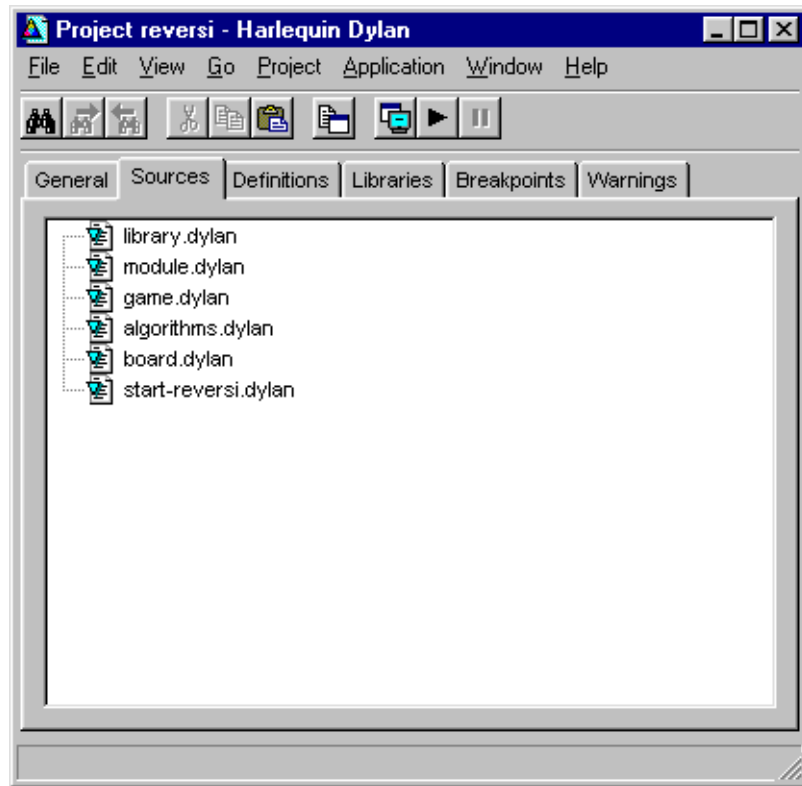


Figure 1.4 The Harlequin Dylan project window.

Projects are a key concept in Harlequin Dylan. Virtually all programming tasks you can carry out in Harlequin Dylan take place in the context of a project. (An exception is editing text files: you can edit a text file without its being part of a project.)

Every time you open a project, or create a new one, Harlequin Dylan displays it in a project window. If you have more than one project open at a time, each is displayed in a separate window.

You will use the project window frequently during development. Some of the things you can do from the project window are:

- Build, run, and start an application.



- Edit source files.
- Browse compilation errors generated when the project was last compiled.
- Browse classes, functions, variables, and so on that the project defines.

In short, the project window is a focal point for work in Harlequin Dylan.

## 1.3 The Reversi project

The project window that is now on your screen has several pages of information about the project Reversi. This project represents a Dylan library called Reversi that can be built into an executable application.

The default view in the project window is the Sources page, which displays the source files that the project contains. We can see that the Reversi project contains six source files. These files include definition files for the Reversi library and the modules it exports — `library.dylan` and `module.dylan` respectively. Later, we will add some more files to the project that implement extra game features.

## 1.4 Building an executable application

We can use the project window to build an executable application from the Reversi project.

This will be a normal Windows executable file with a `.exe` suffix. Harlequin Dylan places the files that it builds in appropriate subfolders of your top-level Harlequin Dylan installation folder. We will see more about this in Chapter 4, “Creating Projects”.

### 1.4.1 Preparing the compiler for our example

Before we can build the project we have to change a default Harlequin Dylan compilation setting. Recall from Section 1.3 that we will be adding some more source files to the project later on. These files implement new game features that require debugging and will help us to demonstrate the Harlequin Dylan development environment further.

When we build the project as it is now, without the extra files, the compiler will detect that some code in the project refers to undefined bindings. These undefined bindings are definitions from the source files yet to be added. When the compiler finds these references, it will stop compiling the project before it gets to its “link” phase, the phase where it would normally create the executable file. Instead, it will record the problems in the project window’s Warnings pane, and stop.

Many programming languages enforce this sort of behavior. In those languages, you can never link and execute a program containing such errors. But sometimes it is useful to be able to execute a program that is only partly finished. As long as we know that the code containing the references to the undefined bindings is not going to be executed, we can safely test the rest of the program.

Reversi has been carefully coded to avoid calling these undefined names until the files containing their definitions are included in the project and the project is rebuilt. So there is no harm in building an executable for Reversi — and fortunately, we can tell the Harlequin Dylan compiler to do so. We simply need to change the default project warnings options in the main window.

1. In the main window, choose **Options > Environment**.  
The Environment Options dialog appears.
2. Choose the Build page.
3. In the Warnings options section, change the setting from “Don’t link if there are serious warnings” to “Always link, even if there are serious warnings”.  
We discuss warnings and serious warnings in “Problems at compile time” on page 20.
4. Click **OK** to confirm the new setting.

We can now go ahead and build an executable Reversi application.

### 1.4.2 Building Reversi

To build the executable Reversi application, go to the Reversi project window.

1. Choose **Project > Build**.

Harlequin Dylan starts building the application. A progress indicator window appears.

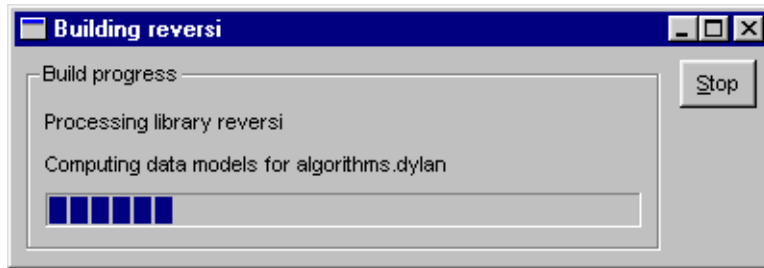


Figure 1.5 The build progress indicator.

Because we have never built the Reversi application before, Harlequin Dylan examines and compiles every source file. It then links the compiled files together with the system libraries that the application uses, and creates an executable file.

### 1.4.3 Running Reversi

Once the application is built, we can run it. The project window menu command **Application > Start** runs the most recently built executable for that window's project.

1. Choose **Application > Start**.

The Reversi window appears.

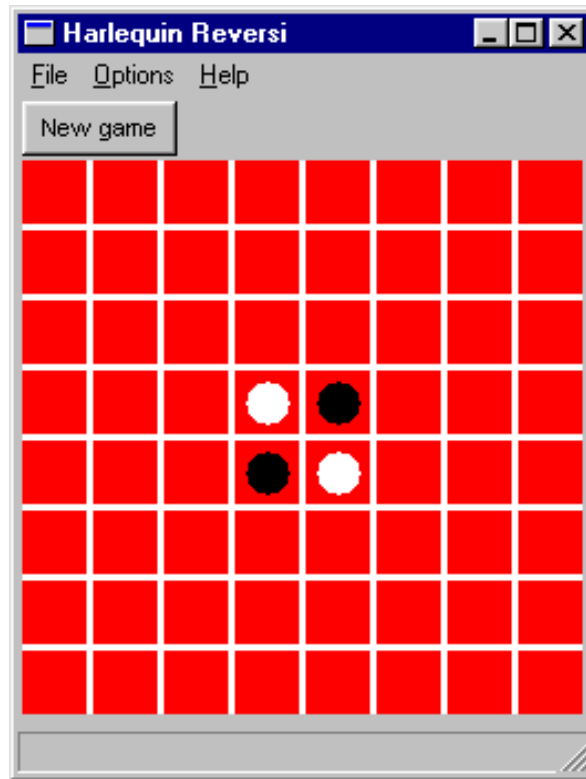



Figure 1.6 The Reversi application.

The Reversi application is now up and running.

When you choose **Application > Start**, the executable starts, runs, and terminates normally, but it is actually running connected to the Harlequin Dylan debugger. This means you can pause execution at any time to debug the application and even interact with it. (Use **Application > Pause** to do this.) In addition, if there is an unhandled error inside the application, the debugger will catch it automatically so that you can examine it. We will learn more about the debugger in Chapter 6.

## 1.5 Three ways of running Dylan applications

There are three ways we can run the Reversi application we have just built. The first is to choose **Application > Start** from the Reversi project window, as we have just seen. This menu command is also available in the Harlequin Dylan debugger and editor.

The second way to run the application is to click the “Start/Resume” button () on the project window’s toolbar. Again, we can do this in the Harlequin Dylan debugger and editor too. And again like **Application > Start**, running an application this way connects it to the debugger, so that any unhandled errors are caught and you can pause and interact with the application.


Finally, you can run the application from outside Harlequin Dylan as you would any other executable application file — such as by typing its file name into an MS-DOS console window, or double-clicking on it in an Explorer window. If you run an application this way, Harlequin Dylan cannot connect a debugger to it.

## 1.6 Looking at definitions

Each file listed on the Sources page of the Reversi project window is now preceded by a + symbol. We can now expand a file name to reveal a list of all the Dylan definitions in that file.

2. Expand the `algorithms.dylan` item by clicking the + next to it.

The definitions are sorted alphabetically under the file name in which they were defined. To be precise, the definitions are the name bindings created by compiling the source code definitions that the file contains.

In addition to the normal Dylan syntactic conventions — a leading dollar sign for a constant, enclosure in angle brackets for a class, and so on — icons appear next to definition names to indicate the kind of Dylan entity to which the names are bound. Constants, for example, are indicated by a circular icon containing a stylized dollar sign (.

You can also see the definitions listed by library and module on the Definitions page. This page includes a facility for filtering definitions out of the visible list according to their type or whether their name contains a given string.

This list of definitions is just one use to which the Harlequin Dylan environment puts its *compiler database*: a rich database of information that the compiler derives from every project you build in Harlequin Dylan.

The fact that the compiler database is derived during compilation explains why the file names in the Sources page were not expandable when we first opened the Reversi project, and also why we would have seen that the Definitions page was empty at that time. The compiler database for Reversi did not exist until after we built the Reversi application. However, when we open the Reversi project in future sessions, Harlequin Dylan will read in the database from disk.

Compiler databases are mostly used by the *browser* tool, which we will look at later. See “Compiler databases” on page 43 for more details.

## 1.7 Making changes

We will now make a change to the Reversi application. We are going to add a new feature that allows someone playing Reversi to change the shape of the pieces.

If you look at the Reversi application again now, you will see that some of the commands on the **Options** menu — Circles, Squares and Triangles — are unavailable. Our changes will enable these items.

Among the Reversi example files, there is a prepared Dylan source file with the changes we need for the new piece-shapes code. It is not yet a part of the project, so to incorporate it into our Reversi application, we must add it to the project.

1. Exit the Reversi application by selecting **File > Exit**.  
You can also use **Application > Stop** to exit running applications.
2. In the Reversi project window, select the Sources tab.
3. Choose **Project > Insert File**.  
The Insert File into Project dialog appears.
4. In the **Files of type** pop-up menu, choose **Dylan Source Files**.
5. Select `piece-shapes.dylan` and click **Open**.

Harlequin Dylan adds `piece-shapes.dylan` to the end of the list of project sources.

The last file in the list of sources starts the main loop of the Reversi application, so `piece-shapes.dylan` needs to come before it in the list. We can use the project window commands **Project > Move File Up** and **Project > Move File Down** to move files in the list.

6. Select `piece-shapes.dylan` in the project window, and choose **Project > Move File Up** to move it above `start-Reversi.dylan`.

Now that `piece-shapes.dylan` is part of the sources that will be used to build the Reversi application, and is in the right place, we can rebuild `Reversi.exe`.

7. Choose **Project > Build** in the project window.

Harlequin Dylan builds the application again.

This time, Harlequin Dylan compiles only one file: `piece-shapes.dylan`. No changes had been made to the existing source files, so it did not need to recompile them. It simply linked the existing compiled files with the new one to make the new executable.

This incremental compilation feature can save a lot of time during development, when you want to rebuild your application after a small change in order to test its effects. Harlequin Dylan automatically works out which files it needs to recompile and which it does not. The compiler also updates a project's database during incremental compilation.

We can now run the new version of Reversi.

8. Choose **Application > Start** in the Reversi project window.

A new Reversi application window appears.

9. In the Reversi application, select the **Options** menu.

Thanks to our compiling the changes to the project, the Circles, Squares, and Triangles items are now available:

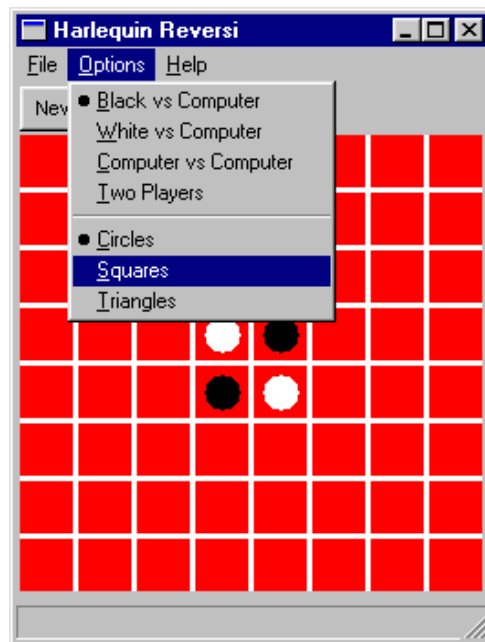


Figure 1.7 The Reversi application's **Options** menu after the code changes.

**10. Choose Squares.**

The Reversi application updates the board, laying the counters out again as squares rather than circles.



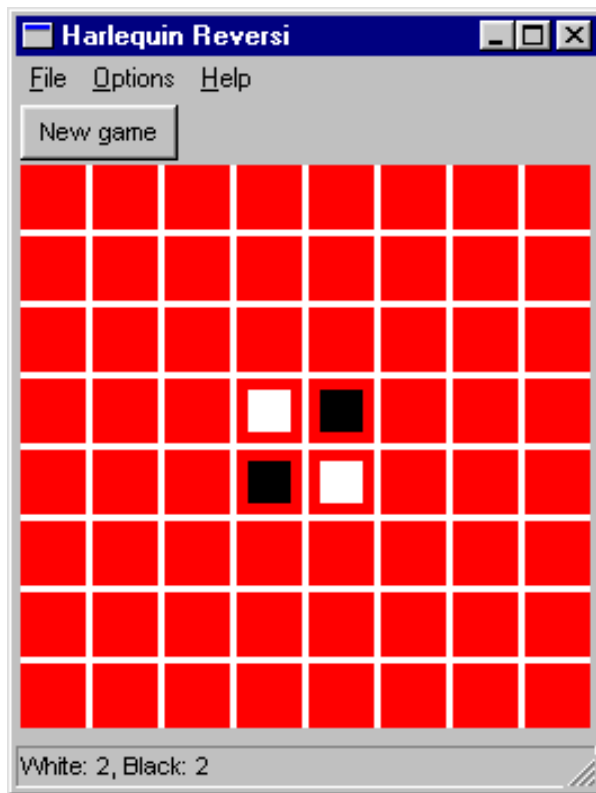


Figure 1.8 The Reversi application with square counters.

## 1.8 Summary

In this chapter, we saw that:

- Harlequin Dylan has a main window that stays on screen throughout your Dylan development session. From the main window, you can set options that affect all windows in the environment.
- Harlequin Dylan is a project-based development environment. Nearly all work in Harlequin Dylan takes place in the context of a project.

- The project window provides the focus for work on a project. You can use the project window to build, run, debug, and interact with applications.
- Harlequin Dylan derives a database of information about a project when compiling it. This compiler database is used in the project window to provide a list of definitions created by each source file, as well as in other tools.
- The compiler database for a project is kept up to date with each compilation. You can save the database to disk so that it will be available for use in future programming sessions.
- Harlequin Dylan provides an incremental compilation facility.
- If you test an application by running it from within the Harlequin Dylan development environment, you can stop it at any time during its execution in order to debug or interact with it.

# 2

---

---

## Fixing Bugs

Though we can play Reversi, we cannot save games in progress and resume play at a later date. In this section, we add a save feature to the game, available through **Save**, **Save As** and **Open** commands on its **File** menu. (If you look at the version of Reversi we have at the moment, you will see that those items are disabled.)

The code we will add to implement the save feature has some small bugs that we must correct before the feature will work properly. In doing so, we will see the editor and debugger.

### 2.1 Rebuilding the application

There is another prepared Dylan source file with the changes we need to implement for the game-saving facility.

1. Choose **Project > Insert File**.

The Insert File into Project dialog appears.

2. In the dialog, select `saving.dylan` and click **Open**.

Harlequin Dylan adds `saving.dylan` to end of the list of project sources.

As we did with `pieces-shapes.dylan` in the first chapter, we must move `saving.dylan` above `start-reversi.dylan`, which starts the application's main execution loop.

3. Select `saving.dylan` in the Sources list and move it above `start-reversi.dylan` with **Project > Move File Up**.

Now we can rebuild the executable Reversi application.

4. Choose **Project > Build** in the project window.

Harlequin Dylan builds the application again.

Notice the message in the status bar at the bottom of the Reversi project window after rebuilding Reversi:

```
Build completed. 5 serious warnings, 4 warnings.
```

In the next section, we look at what serious warnings and warnings are, and how the environment helps us deal with them.

## 2.2 Problems at compile time

In this section we look at how the compiler handles problems that it comes across in a project's source code. When we rebuilt Reversi in Section 2.1, the compiler discovered nine problems in the project's source code. It reported those problems with five *serious warnings* and four *warnings*.

The compiler issues a *serious warning* for code that would lead to an exception at run time if it was executed. Among the causes of serious warnings are Dylan syntax errors, references to undefined bindings, and calls to functions with the wrong number or type of arguments.

For code with only cosmetic problems, such as a method definition that ends with `end class` instead of `end method`, the compiler issues a *warning*.

You can see a table of any warnings or serious warnings that were generated the last time a project was compiled by choosing the Warnings page in the project window.

1. Select the Warnings page in the project window.

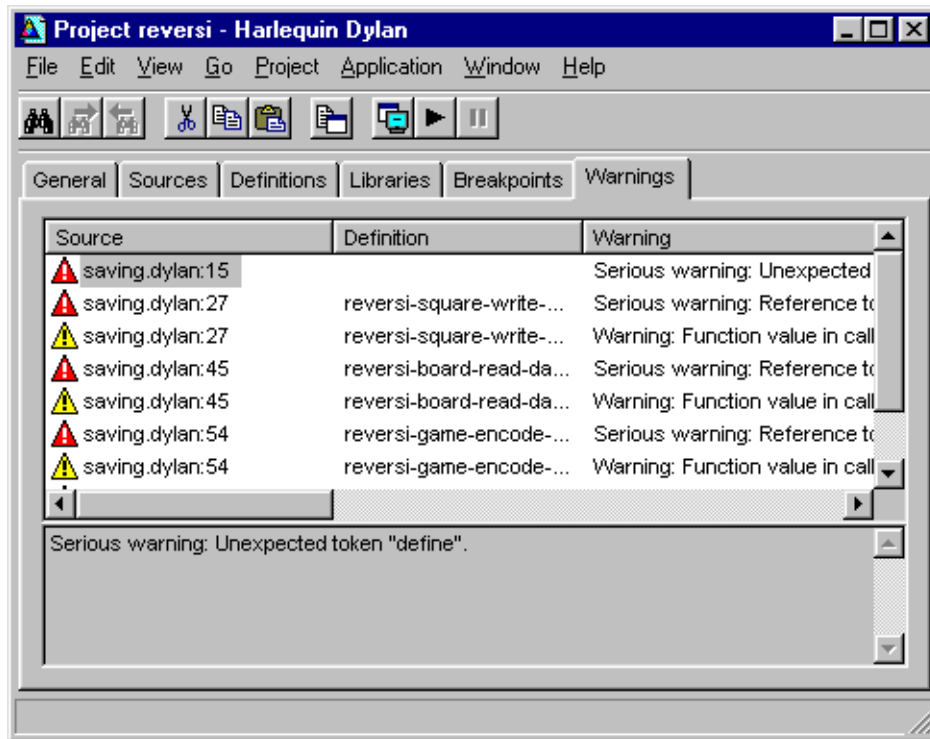


Figure 2.1 The project window's Warnings page.

The table has three columns:

Source	Shows the source code location of the problem that caused the warning or serious warning. The column has the form { <i>file name</i> }:{ <i>line number</i> }.
Warning	Shows the warning or serious warning message.
Definition	Shows the source code definition that caused the warning or serious warning, if any. This column is left blank if the problem is not associated with a definition.

The items in the table are sorted lexically by the Source column value. Warnings and serious warnings without associated source locations have empty

Source fields, and appear at the top of the list. (To sort the table by the values in another column, click on that column's header.)

Each warning, serious warning, or error is linked to the source definition that caused it. We can select individual warnings, serious warnings, and error messages to learn more about the problem they are describing.

2. Click on the first item in Source column, a serious warning in `saving.dylan` at line .

We can now see the full text of the serious warning:

```
 Serious warning: Unexpected token "define".
```

If we double-click on an item, Harlequin Dylan opens the source file in an editor window, at the line that caused the problem.

3. Double-click on the same item.

An editor window appears.

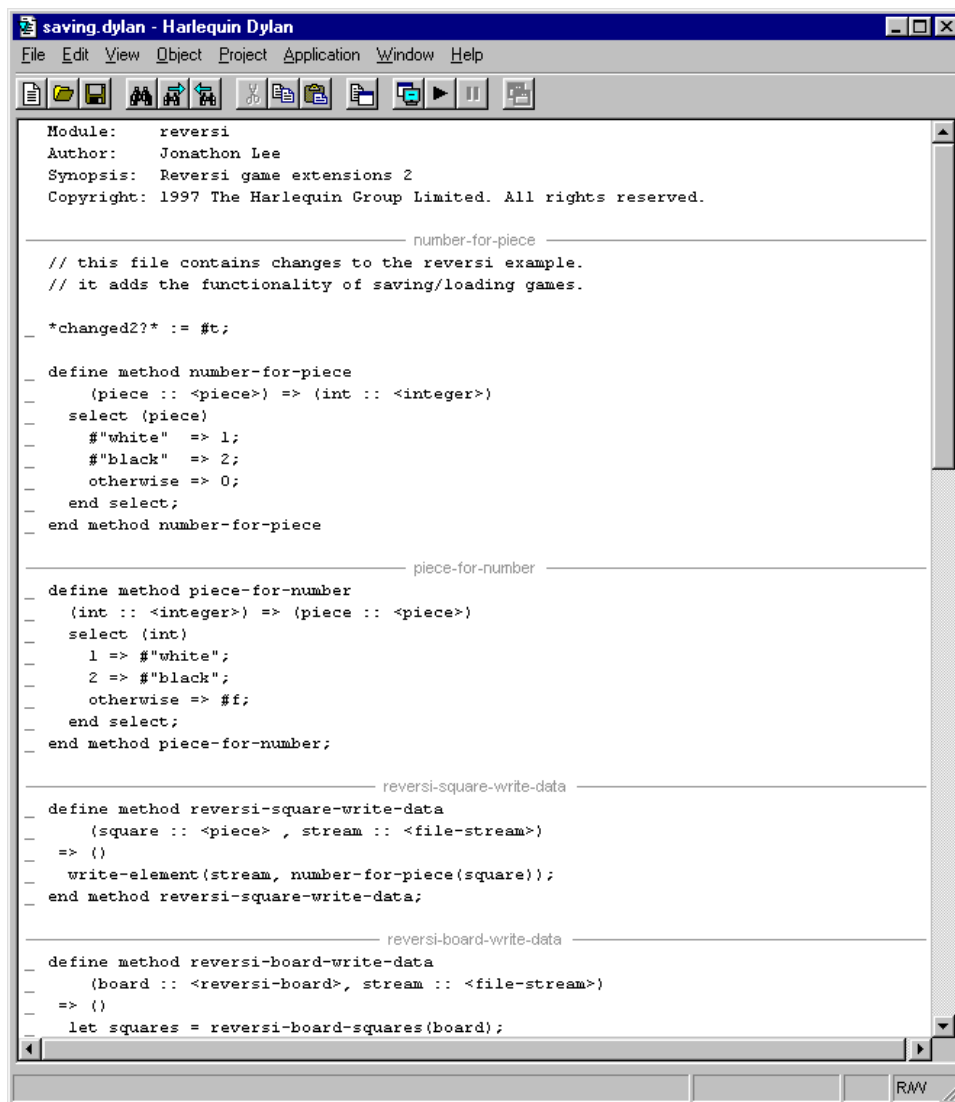


Figure 2.2 The Harlequin Dylan editor.

Now we can see the cause of the first serious warning. A semi-colon is missing from the end of the definition of `number-for-piece` in `saving.dylan`. The

missing semi-colon makes the definitions of `number-for-piece` and `piece-for-number` run together.

As it turns out, all the other serious warnings and warnings that were reported were caused by this single missing semi-colon. The compiler could not parse the definitions of `number-for-piece` and `piece-for-number`. The compiler skips over such source code and does not generate object code for it.

This means that any subsequent references to `number-for-piece` and `piece-for-number` in the source code would be references to name bindings that are never defined in the compiled application. Lines , , , and referred to one or the other of these names, which triggered the warnings and serious warnings.

4. Add the missing semi-colon so that the last line of the definition `number-for-piece` appears as follows:

```
end method number-for-piece;
```

5. Choose **File > Save** in the editor.

Harlequin Dylan saves the file, first making a backup of the previous version in `saving.dylan.bak` — that is, in a file of the same name, but with an extra extension, `.bak`, to show that it is a backup file.

Having attended to the cause of the warning, we can rebuild the application and try out the new version.

6. Choose **Project > Build**.

You can choose this in either the editor or the project window.

Notice the status bar in the Reversi project window after the build is complete:

```
Build completed with no warnings.
```

As well as the serious warning our correction addressed, it has removed all the other warnings and serious warnings that were knock-on effects of the missing semi-colon.

## 2.2.1 Controlling the compiler's treatment of warnings

We have seen that serious warnings are caused by code that, if executed, would lead to a run-time exception. Some programming language compilers



would refuse to create (“link”) an executable file for projects containing such code, on the grounds that the code ought to be fixed before it is executed.

In Harlequin Dylan, you can choose whether to create an executable or DLL for a project containing such code. The choice is controlled from the main window, under the Build page of the **Options > Environment Options** dialog.

In Section 1.4.1 on page 9 we changed this setting, to make the compiler link an executable file even if there were serious warnings. We changed the Warnings options on the Build page to “Always link, even if there are serious warnings”, forcing the compiler to link an executable file for Reversi despite any serious warnings. If we had not done this we would not have been able to run Reversi without `piece-shapes.dylan` and `saving.dylan` incorporated first.

The other possibilities are “Don’t link if there are serious warnings”, and “Ask whether to link if there are serious warnings”.

In addition, we can check a box on the Build page to make the compiler treat all warnings as serious warnings. Turning this option on makes the compilation process extremely strict: even a cosmetic error will stop the compiler linking an executable file or DLL for the project.

## 2.3 Problems at run time

Now we have taken a brief look at how Harlequin Dylan treats compile-time problems, we will look at how it lets us debug problems that only emerge as exceptions at run time.

**Note:** The numbered example steps in this section lead us through a possible debugging scenario. In places the example is a bit unrealistic, because usually you are familiar with at least some of the code you are debugging, and also because the main purpose of the example is to introduce features of Harlequin Dylan.

1. With the rebuilt version of Reversi that compiled with no warnings, start a new game, with **Application > Start**.
2. After a couple of moves, save the new game by choosing **File > Save** in the Reversi window.

The Save dialog appears.

3. Choose a file to save into, and click **OK**.

An unhandled Dylan error dialog appears.

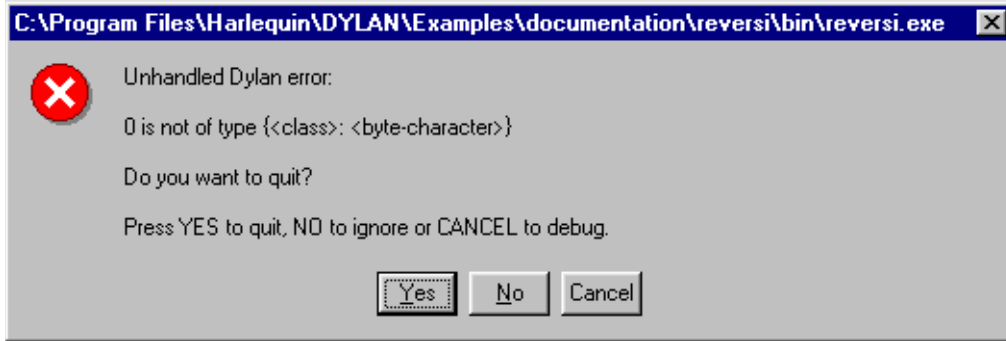


Figure 2.3 An unhandled run-time error.

The dialog appeared because the Harlequin Dylan debugger caught an unhandled exception in the Reversi application. Clearly, something is wrong with the game-saving code. We must start up a debugger window to see what went wrong.

4. Click **Cancel** to enter the debugger.

The Harlequin Dylan debugger appears. We discuss the debugger in detail in Chapter 6, “Debugging and Interactive Development”.

In its uppermost pane, the debugger shows the error that it caught. It will be either:

```
Dylan error: n is not of type {<class>: <BYTE-CHARACTER>}
```

where *n* is either 0, 1, or 2. (The value depends on the state of the game when we saved it. The reason for this will become clear shortly.)

In the left-hand pane beneath the message, there is a tree item for the master thread of the Reversi application. This tells us that the exception was raised in that thread. (In Reversi’s case, there happens to be only one thread, but other applications might have multiple threads, and knowing the thread that raised the exception is useful. In a multi-threaded application, each thread being debugged will have its own debugger window.)

When expanded, the tree item shows the current state of the call stack for Reversi's master thread. When the debugger is invoked on a thread, it pauses execution in that thread. So when we expand the tree we see the stack exactly as it was at the moment that the debugger was invoked.

5. Click on the cross to expand the master thread stack details.

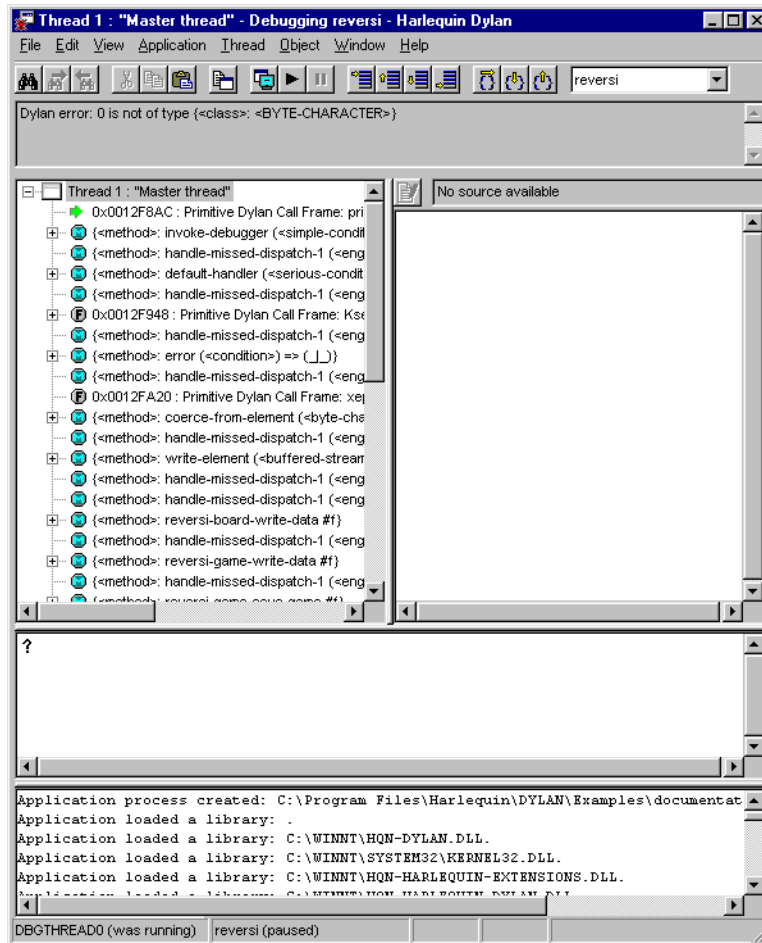


Figure 2.4 The Reversi application stack after a game-saving error.

Each item in the expanded list is a call frame on the stack for the thread being debugged. We call this list of call frames a stack backtrace or simply a *backtrace*. The backtrace shows frames in the order they were called, with the most recent at the top of the list.

### 2.3.1 Searching the stack backtrace for the cause of the error


In this section we examine the backtrace and see what events led up to the unhandled exception.

Looking at the top of the backtrace, we can see that the most recent call activity in the Reversi master thread concerned catching the unhandled exception and invoking the debugger. The calls to `invoke-debugger`, `handle-missed-dispatch-1`, `default-handler` and `error` were all part of this. But if we move down the backtrace to the point below the call to `error`, we can examine the sequence of calls that led to the unhandled exception and find out how to fix the error.

**Note:** The calls to `handle-missed-dispatch-1` are internal details that we need not consider.

The first interesting call for us is the one to `coerce-from-element`. This was the last call Reversi made before the unhandled exception.

#### 6. Select the call frame for `coerce-from-element`.

The source code definition of `coerce-from-element` appears in the pane opposite. This source code pane is read only; if we wanted to edit a definition shown in it we would click on the Edit Source () button above the source code pane, which would open the file containing the definition in an editor window.

Looking at the source code for `coerce-from-element`, we can see that all it does is call `byte-char-to-byte`. Notice that this call does not appear in the backtrace. That tells us that the exception must have been raised while attempting to call this function.

Since the error dialog told us that the exception was caused by something being of the wrong type, there is a good chance that the value of `elt`, the argument here, is of the wrong type. Notice too that `elt`'s type is not specified in the signature of `coerce-from-element`.

We need to know the value passed to `elt`. We can find out by expanding the `coerce-from-element` call frame: a call frame preceded by a `+` can be expanded to show the values of its arguments and local variables.

7. Expand the call frame for `coerce-from-element`.

We can now see the value that was passed for `elt`. It is an integer value, and not an instance of `<byte-character>`.

Our next task is to find out why `coerce-from-element` was sent an integer instead of a byte character. To do this, we can simply move down the back-trace and examine earlier calls.

8. Select the call frame for `write-element`.

We can see here that the value passed to `elt` in `coerce-from-element` is the value of one `write-element`'s parameters, also called `elt`.

We need to move further down the stack to the `reversi-board-write-data` call. That call frame is displayed like this (without the line-wrap, and with hexadecimal values at the start and the end):

```
Primitive Dylan Call Frame:
Kreversi_board_write_dataYreversiVreversiMreversiM0|
```

9. Select the call frame for `reversi-board-write-data`.

The `reversi-board-write-data` method takes an instance of `<reversi-board>` and an instance of `<file-stream>` as arguments. A `<reversi-board>` instance is what the game uses to represent the state of the board during a game. A `<file-stream>` is what Reversi is using to write the state of the board out into a file that can be re-loaded later.

We can see that this method calls `reversi-board-squares` on the `<reversi-board>` instance and then iterates over the value returned, apparently writing each element to the stream with `reversi-square-write-data`.

We are closing in on the bug. It is looking like the value representing the Reversi board squares (`squares`), and the file stream the squares are being written to (`stream`), have incompatible element types, with the squares being represented by integers, and the file stream being composed of byte characters.

To confirm that the elements of `squares` are integers, we can examine the actual run-time instance — the value of `squares` in the `reversi-board-write-data` call frame — using the debugger’s interaction pane.

### 2.3.2 Browsing local variables

In this section we use the Harlequin Dylan browser to help confirm the cause the unhandled exception.

10. Expand the call frame for `reversi-board-write-data`.

(Recall that it is printed as a Primitive Dylan Call Frame.)

We can now see the values of the local variables in this frame. The arguments are listed first: `board` and `stream`, followed by the iteration variable `square`.

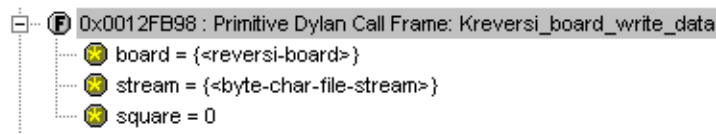


Figure 2.5 Local variables in the `reversi-board-write-data` call frame.

The notation

```
board = {<reversi-board>}
```

means that `board` is an instance of `<reversi-board>` — an actual instance in the paused thread. The curly braces notation tells us that this is an instance of the class rather than the class definition itself.

We can look at the instance in the browser, which allows us to examine the contents and properties of all kinds of things we come across in Harlequin Dylan.

11. Double-click on the `board` item.

The browser appears. We discuss the browser in detail in .

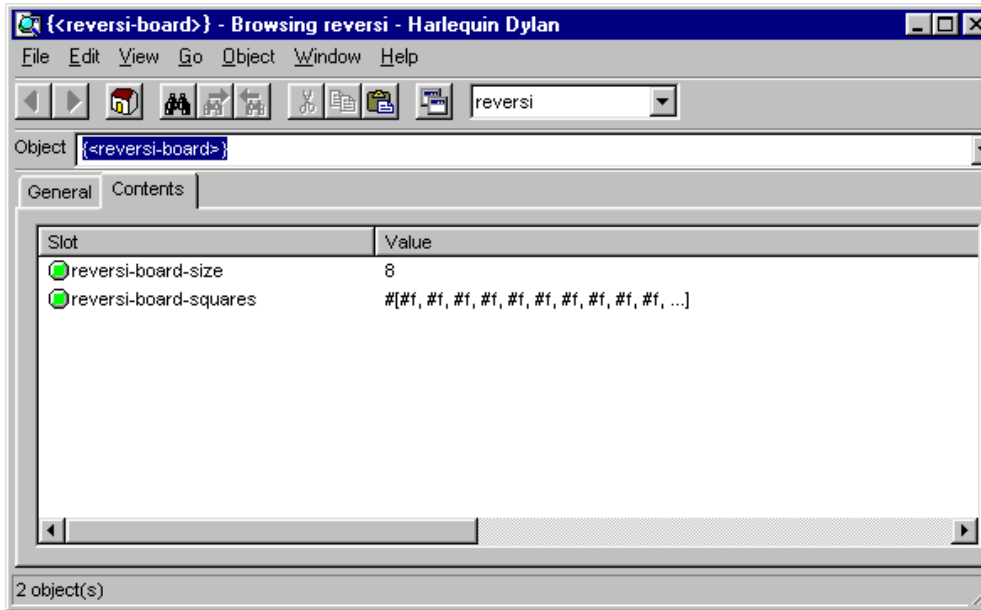


Figure 2.6 Browsing an instance of `<reversi-board>`.

The browser shows us in its Object field that we are browsing an instance of `<reversi-board>`. Like the debugger, the browser uses the curly braces notation to depict an *instance* of a class as opposed to its definition.

The browser presents information in property pages. In the page selected by default, we see the names of the slots in the instance and the values they had when the exception occurred. The property pages that the browser shows depend on what it is browsing; the set of pages for a class definition is quite different from that for a method definition, for example.

However, the browser always provides a General page. The General page gives an overview of the currently browsed object.

## 12. Choose the General page.

The General page for our `<reversi-board>` value tells us that it is an instance and that it has two slots. The other fields are labeled *n/a* for “not applicable”; we will see why in .

Going back to the bug we are tracking down, two more useful pieces of information emerge from seeing this instance in the browser. First, we can tell from the Contents page that the call to `reversi-board-squares` in `reversi-board-write-data`, below, is clearly just a call to the default accessor on the `<reversi-board>` slot of the same name.

```
define method reversi-board-write-data
  (board :: <reversi-board>, stream :: <file-stream>)
  => ()
  let squares = reversi-board-squares(board);
  for (square from 0 below size(squares))
    reversi-square-write-data(squares[square], stream);
  end for;
end method reversi-board-write-data;
```

Second, we can see that `reversi-board-squares` holds a sequence, and that sequence does not have an `<integer>` element type.

So we still do not know where the integer that caused the exception came from. However, we have yet to check what goes on in `reversi-square-write-data`; perhaps that method is converting the elements in the `reversi-board-squares` sequence into integers?

### 2.3.3 Browsing definitions

In this section, we browse the definition of `reversi-square-write-data` to see whether it converts the board squares into integers.

To browse the definition we have the option of locating it on project window Definitions page or, more efficiently, we can move to it directly in the browser.

13. Delete the text in the Object field and type `reversi-square-write-data` in its place.
14. Press Return.

The browser switches to the definition of the `reversi-square-write-data` method. The default property page is the Source page, which shows the source code for the method.



```

define method reversi-square-write-data
  (square :: <piece>, stream :: <file-stream>)
  => ()
  write-element(stream, number-for-piece(square));
end method reversi-square-write-data

```

So `number-for-piece` is most likely returning the integer value that was passed to `write-element` (and that we can see on the stack as the `elt` local variable). The `square` value has type `<piece>` — this, then, is the element type of the sequence used to represent the state of the board.

15. Browse the definition of `number-for-piece`.

The definition of `number-for-piece` completes the story. It is here that the board square representations are converted into integers. This is where the integer that caused the exception came from.

### 2.3.4 Fixing the error

In this section, we fix the Reversi project source code to eliminate the cause of the exception we have been tracking down.

This is what we have learned about the error so far:

- It occurred when trying to save a Reversi game.
- It was caused in a call to `coerce-from-element`, which attempted to pass an integer to `byte-char-to-byte`, a method which expects an instance of `<byte-character>`.
- The `coerce-from-element` method received the integer from `write-element`, which received the integer from `reversi-square-write-data`.
- The `reversi-square-write-data` method uses the `number-for-piece` method to translate board square representations (type `<piece>`) into instances of `<integer>`.

In addition:

- The `write-element` generic function is from the Harlequin Dylan Streams library. It is part of that library's protocol for writing to streams.
- The stack shows that the `write-element` method that was called tried to coerce an integer to a byte character, and that coercion failed.

So we know that Reversi is trying to write integer values to a file stream with a `<byte-character>` element type, and the exception occurs during the attempt to coerce an integer into a byte character.

We could simply change the file stream's element type to `<integer>`.

In fact, we have not yet looked at the call that created the file stream. That call is `reversi-game-save-game`.

16. Return to the debugger and select the call frame for `reversi-game-save-game`.

As expected, the source pane shows that the file stream is created with an element type of `<byte-character>`. The relevant code fragment is:

```
let file-stream = make(<file-stream>, locator: file,
                      direction: #"output",
                      element-type: <byte-character>);
```

17. Click the Edit Source button.

We now have `saving.dylan` in the editor, and the insertion point is positioned at the start of the definition for `reversi-game-save-game`. We can make the change to `<integer>`, but should first check `reversi-game-load-game`, the method that loads games saved by `reversi-game-save-game`, to see what sort of file-stream elements it expects to read back.


That definition is located directly below that of `reversi-game-save-game`. It shows that the element type `reversi-game-load-game` expects is `<byte>`.

```
let file-stream = make(<file-stream>, locator: file,
                      direction: #"input",
                      element-type: <byte>);
```

The class `<byte>` is actually a constant value, defined:

```
define constant <byte> = <integer>;
```

(We can confirm this by entering `<byte>` into a browser window's Object field.) So there is no harm in changing the `element-type:` argument in `reversi-game-save-game`'s call to `make` from `<byte-character>` to `<integer>`, but for symmetry we may as well change it to `<byte>`.

18. Click on the Edit Source () button above the source code pane.

An editor window opens on `saving.dylan`.

19. Fix the definition of `reversi-game-save-game`.

The `element-type:` keyword in the call to `make-on` on `<file-stream>` should take `<byte>`, not `<byte-character>`.

20. Choose **File > Save**.

Before we can rebuild the application we need to stop the current version running.

21. Choose **Application > Stop**.

22. Rebuild the application with **Project > Build**.

23. Start the application again, and try to save a game.

The save operation now works without raising an unhandled exception.

### 2.3.5 Loading the saved game back in

The next step is to test the code for loading a saved game. To test this we need to change the state of the board from what it was like when we saved the game.

24. Clear the Reversi board by clicking **New Game** in the Reversi application.

25. Choose **File > Open** in the Reversi application, select the file, and click **Open**.

Another error dialog appears.

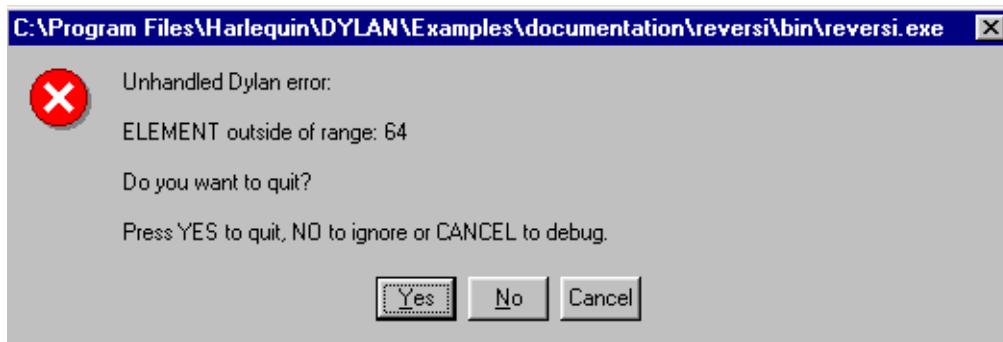



Figure 2.7 Another unhandled run-time error.

26. Choose **Cancel** to enter the debugger.

This second error occurs because the game-saving code reads off the end of the saved game file. Browsing the stack shows that the offending definition is `reversi-board-read-data`, which should, like its opposite `reversi-board-write-data`, use the `for ... below` idiom, not `for ... to`, in its loop through board squares from zero up to `size(squares)`.

27. Select `reversi-board-read-data` in the stack backtrace.

28. Click on the Edit Source () button above the source code pane.

An editor window opens on `saving.dylan`.

29. Fix the definition `reversi-board-read-data`.

The loop should use the `for ... below` idiom, not `for ... to`.

30. Save the source file and rebuild the Reversi application. You will need to stop the running version of Reversi to do this.

31. Choose **Application > Start**, and load the saved game file in.

Reversi now shows the state of the game you saved earlier.

## 2.4 Summary

This section yet to be written.

# 3

---

## Programming in Harlequin Dylan

Now we have taken a tour of Harlequin Dylan using the pre-written Reversi application, we will take a step back to look at the programming model in Harlequin Dylan, and review the features of the development environment and the Dylan compiler.

### 3.1 Projects

In Harlequin Dylan, you do all development work in terms of *projects*. Projects are the development environment and compiler's way of representing Dylan libraries.

A project consists mainly of a list of the source files that define the library, but also contains information about how to compile the library.

While it is possible to edit text files that are not associated with any project, nearly all other programming tasks in Harlequin Dylan take place within the context of a project.

#### 3.1.1 Projects and libraries

A project represents a single Dylan library. Think of a project as something that gathers together all the information Harlequin Dylan needs to be able to compile a particular library.

For example, earlier in this manual, we worked with the Reversi project. The Reversi project represents a single Dylan library, called Reversi.

### 3.1.2 Projects and deliverables

You can create deliverable applications, libraries, and components from projects. Projects can be built into executable (.EXE) or dynamic-link library (.DLL) files.

When we worked with the Reversi project, we built an executable from it, but we could just as easily have built a DLL. See “Project settings” on page 61 for details.

### 3.1.3 Creating new projects

Harlequin Dylan includes a New Project wizard for creating projects.

The wizard asks questions about what you are going to develop, and then creates a new project configured to help you get working quickly.

You can also create a project if you have a Dylan Library Interchange Description (LID) file for it. When you open a LID file in the development environment, it is converted into a project file and opened in a project window. (This process does not modify the original LID file on disk.)

### 3.1.4 Project files

Harlequin Dylan stores projects on disk in *project files*. Project files have the same name as the project, but with the extension .HDP. Thus a project called Hello is stored in the file `Hello.hdp`. (Projects get their names in the New Project wizard or, if they were created by conversion from a LID file, get their name from the name of library that the LID file defines.)

Project files are the files you select when you want to open a project or add it to another project as a subproject.

The development environment tools automatically save changes to project files. For example, if you add a new source file to a project, the change is saved to disk immediately. Each time Harlequin Dylan saves a project file, it makes a backup file with a .BAK suffix. Thus the backup file for `Hello.hdp` would be called `Hello.hdp.BAK`.

### 3.1.5 Project components

We now describe the components of a project more closely.

A project consists of:

1. A Dylan library

Every project defines a single Dylan library. We call this library the “library of the project” or, for clarity, the *main* library of the project, to distinguish it from other libraries that the project uses.

2. A project name

Every project has a name. When you create a new project with the New Project wizard, the wizard uses this name to generate default names for initial files, libraries and modules in the new project. The compiler uses the project name to generate default names for the executables and other files it produces during compilation. In both cases, you can override the defaults.

3. Source code files, and other files

Every project includes source code files. Projects created with the New Project wizard will have a `library.dylan` file (which defines the project’s library); a `module.dylan` file (which defines the modules of the library); and at least one other Dylan source code file containing definitions and expressions.

Projects can also include Windows resource files and static libraries (.LIB files).

4. Subprojects

A project can have *subprojects*. Subprojects are simply other projects that are included in a project; they define their own main library, contain their own source files and may have subprojects themselves. For clarity, we can call a project a *superproject* when describing it with reference to its subprojects.

See “The build cycle” on page 45 for more on the relationship between projects and their subprojects.

## 5. Version numbers

Every project has a major and minor version number. The major version number affects the build dependencies across projects, as described in “The build cycle” on page 45.

## 6. Project settings

Every project has settings. Among these settings are:

- The list of source code files and their locations on disk.
- Compilation mode options. See “Compilation modes” on page 44.
- Debugging options. See .
- The list of subprojects the project uses.
- The locations on disk of the subprojects.
- The list of breakpoints set in the project, its subprojects, and the libraries it uses.

### 3.1.6 Projects on disk

As already stated, a project consists of a number of files. All projects are represented with a project file (.HDP file), and they contain Dylan source files and possibly Windows resource and static library (.LIB) files.

The files that make up a project are stored in a disk folder with certain subfolders. The files themselves may refer to other folders where subprojects and used libraries are stored.

The project folder on disk will contain the following files and subfolders:

1. The project file. (.HDP file.)
2. The source code files. (.DYLAN files)
3. The `bin` folder.

This folder holds the executable (.EXE) or DLL (.DLL) file produced from the project.

In addition, the DLLs of the project’s subprojects are automatically copied into this directory, so that they will be on the *DLL search path* when you execute your project’s application.



#### 4. The `build` folder.

This folder holds a number of intermediate files produced during builds. You will never have to do anything with these intermediate files.

The folder also contains the *compiler database* file for the project. This file has the same name as the project and the extension `.DDB`. See for more details.

#### 5. The `lib` folder.

This folder holds the *linker* file for the project. This file has the same name as the project and the extension `.LIB` or `.DEFS`. You need this file to link against the project (which is part of using it as a subproject).

The extension is `.LIB` if you are using the Microsoft linker, or `.DEFS` if you are using the GNU linker.

### 3.1.7 Projects in the development environment

The Harlequin Dylan development environment offers a variety of ways to examine and manipulate projects. You can view a single project in multiple windows at the same time. You can also have more than one project open in the environment at a time.

Apart from the main window and dialog boxes, windows in Harlequin Dylan are generally instances of programming tools. The tools provide views onto different pieces of a project, or sometimes different views of the same pieces.

For example, you might want to have editor windows open on multiple files in the project, as well as browser windows to show you structural views and debugger windows to show you stack backtraces or other information from a running program.

As we saw when touring the environment with the Reversi example, Harlequin Dylan offers:

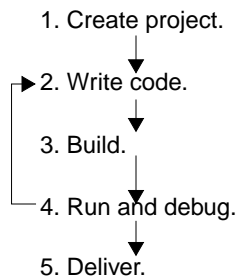
- A project window.
- A debugger for examining and interacting with paused application threads associated with open Dylan projects.

- A browser for examining the contents and properties of projects and of the objects in paused application threads associated with open Dylan projects.
- An editor for source files. Editors are most often invoked from other windows on a project, but can be invoked on files outside the context of a project.

## 3.2 Development models

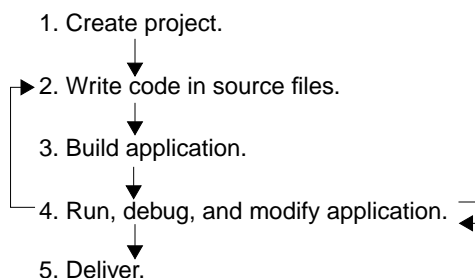
This section explains the development process in Harlequin Dylan.

The process of development in Harlequin Dylan can be much the same as in interactive development environments for other languages. Applications written in Dylan can be developed in the same way as applications written in static languages like C and C++, for instance.



**Figure 3.1** “Static” development model.

You can also develop applications in a more dynamic fashion, using features in the debugger and browser tools that allow you to interact with a running application. With these dynamic, interactive features, you can test bug fixes on the fly and keep your application running before committing to a rebuild.



**Figure 3.2** “Dynamic” development model.

### 3.2.1 Interactive and incremental development

Harlequin Dylan offers both interactive and incremental development features. It is important to distinguish them clearly:

Incremental development is the ability to recompile portions of a project and save the resulting object code. By contrast, some compilation systems require that the entire project be recompiled in response to any change, however small. Harlequin Dylan always performs incremental compilations when it can, to keep build times as short as possible.

Interactive development is the ability to execute code fragments, including definitions and redefinitions, in a running program. Harlequin Dylan offers interactive development via the debugger’s interaction pane. In general, the object code produced during interactive development is not saved, but just patched into the running program and added to the in-memory *compiler database* (see page 44). The object code is lost when the program terminates.

## 3.3 Compilation

### 3.3.1 Compiler databases

When compiling a project, Harlequin Dylan produces a compiler database which models the project. The database provides a rich source of information to Harlequin Dylan tools about the contents, properties, and relationships between source code definitions, libraries, and modules.

A project's compiler database is used when browsing and debugging the project, and is also used when compiling other projects that use the project.

The compiler database for a project does not exist until the project has been built for the first time. Before then, if you try to do anything that requires the database, the development environment will ask you if you want to create it.

Once the compiler database has been built, the development environment will ensure it is kept up to date with each recompilation of the project.

Compiler databases are created and used in RAM, but can be stored on disk for persistence between Harlequin Dylan sessions. When you close a project, the development environment checks whether the database has changed since it was last saved, and if it has it asks you if you want to save the database. When you re-open the project later, the database is read into memory from the disk file, if it exists.

Compiler database files have a .DDB suffix.

### 3.3.2 Compilation modes

The Dylan language encourages programmers to write programs that can be compiled as efficiently as programs written in static languages. By adding type declarations and sealing to your project code, the Harlequin Dylan compiler can optimize it very successfully.

However, the best optimizations come at the costs of longer build times, and less symbolic information in the debugger. During the larger proportion of your project's development, you want projects to build quickly and to be easier to debug. When it is time to deliver your product, you will want to turn all the code optimizations on even at the expense of debugging information and compilation speed.

Like other compilation systems, Harlequin Dylan allows you to switch between both styles of compilation. For any project, you can specify the style of compilation to perform by choosing **Project > Settings** in any window with a **Project** menu, and then choosing the Compile property page.

That page offers two mode choices:

- Interactive Development mode

- Production mode

You should do the majority of your work on a project in Interactive Development mode. When compiling a project in this mode, the compiler does not perform as many optimizations as it can, and is not as strict about error checking as it can be. The idea here is to keep compilation times as short as possible.

This mode keeps symbolic information in the compiled code that will make debugging work easier. Also, if your project was compiled in this mode you will be able to do more interactive work in the debugger's interaction pane, including redefinition. However, compiled code will not be as fast as it can be.

When your project work is nearing completion, and you want to see the compiled version running as fast as possible, switch to compiling the project in Production mode. Production mode turns on all compiler optimizations. However, build times will be slower than in Interactive Development mode, and debugging and interaction will be more limited.

When you have switched to production mode, you can use Harlequin Dylan's *optimization coloring* feature to highlight inefficiencies in your code. This feature colors source code so that you can see where optimizations did and did not occur. Adding type declarations and sealing will secure new optimizations, which you can verify by refreshing the coloring after rebuilding the project. See

### 3.3.3 Binding

Interactive Development mode and Production mode are in fact combinations of some lower-level compiler settings.

This section yet to be written. Will include:

- Loose/tight binding and implications
- How compilation modes relative to loose/tight

### 3.3.4 The build cycle

Building an application or DLL from a project consists of up to three phases:

1. Building the subprojects.
2. Compiling some or all of the project source code.

### 3. Linking the project.

For efficiency, when the compiler is asked to build a project it minimizes the number of these phases that it performs, using the following decision rules:

- If phase 2 or 3 is performed, the project is considered changed.
- A *clean build* always performs all phases for the project and its sub-projects.  
You can ask for a clean build by choosing **Project > Clean Build** in any window that has a **Project** menu.
- A build command is always recursively performed on subprojects (phase 1).
- If the major version number of any subproject has been changed, then all of the source code in the project is recompiled.
- If the project is tightly bound to any subproject which has changed, then all the source code in the project is recompiled.
- If the project is tightly bound to itself, and if any source code in the project has changed, then all the source code in the project is recompiled.
- If the project is loosely bound to itself, then any source code files that have changed are recompiled. Additionally, files that depend on those changes (such as through macro usage) are recompiled.
- If the project or any of its subprojects has changed, then the project is relinked.

**Note:** To ensure change propogation according to these rules, you should always increment the major version number of a project after altering any macro definitions in it.

## 3.4 Executing programs

This section yet to be written. Will include:

- DLL search path
- Library initialization

- Starting applications up from within Harlequin Dylan

## 3.5 Source, database, and run-time views

We have seen that Harlequin Dylan provides several tools to allow us to view projects in different ways. Some tools can look at the source representation of a project, while others can look at the run-time representation — the threads of a running application built from a project.

In fact there are three parallel “universes” in which we can view projects: source, database, and run-time.

Every project has a representation in source code. When you build a project, the compiler database that is created provides a second representation. Then, when you run the application or DLL you have created, the running program is a third representation of the project.

So, at any given time, an object may exist in each of these universes simultaneously. The source code of the object may exist in a Dylan source file, a model of the object may exist in the compiler database, and the object may be instantiated in a running program.

Editor windows show projects in their source representation only. Browser windows show information from the compiler database, and, if a program is running, this database information is combined with information from the program, so you can see the “live” version of the object.

The debugger and its interaction pane allows you to view the threads of running programs, and allows you to execute expressions and definitions in these threads. When you do this, the running program is modified. When you enter definitions in this way, the definitions are saved in a temporary layer of the compiler database so that browsing will continue to be accurate. However, these temporary changes are not saved to disk in the compiler database file, nor are they reflected in the project source code files.

There are ways in which the three universes can get out of sync. Remember that if you edit a source code definition, the model of it in the database will not be updated until you rebuild the project. So, for instance, if you change the inheritance characteristics of a class, the change will not be reflected in the

browser Superclasses page for that definition until you rebuild. And if you add new definitions to the project sources, they will also not be visible until you build the project again.

## **3.6 Summary**

This section yet to be written.



# 4

---

---

## Creating Projects

In this chapter we look more closely at projects. We take a break from the Reversi application and develop a new application of our own. To do this, we create a new Harlequin Dylan project with the New Project wizard.

### 4.1 Creating a new project

We now create a new project for a simple “Hello World” application. We can do this by clicking the New Project button in the main window, or by choosing **File > New** in any window.

1. Select **File > New** from the project window.

A dialog appears, offering the choice of creating a new text file or a new project file.

2. Select “Project File” and click **OK**.

The New Project wizard appears.

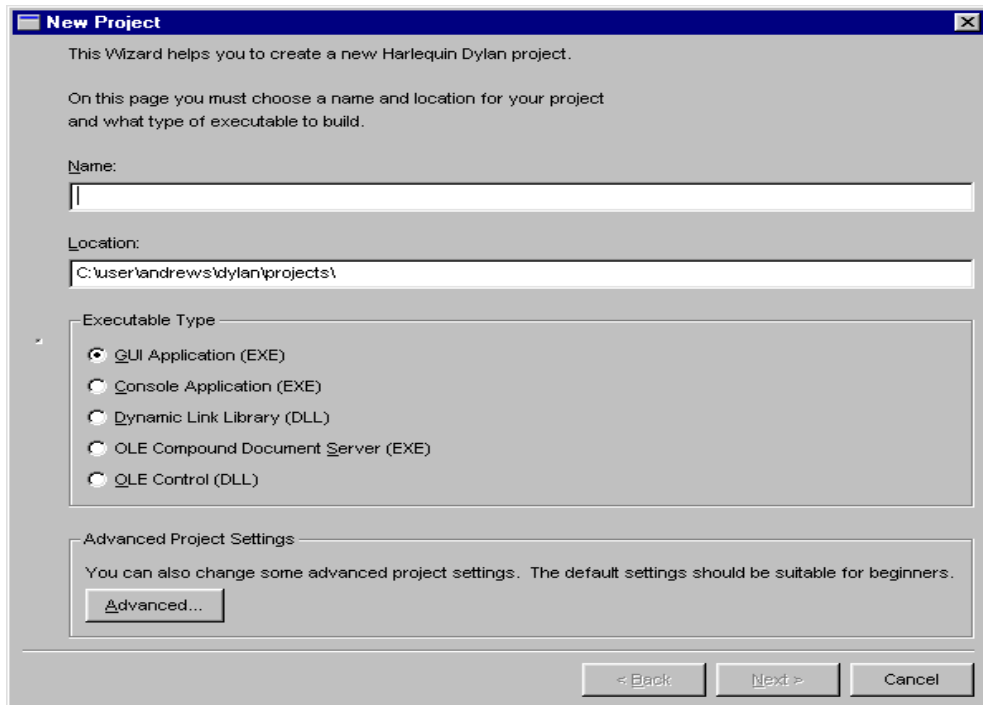


Figure 4.1 The New Project wizard.

The New Project wizard guides us through the process of creating a new project. When we have supplied all the information it requires, the wizard creates a new project, consisting of the following files:

- A project file. The file has the same name as the project, but with a `.HDP` extension.
- A Dylan source file, called `library.dylan`, that defines a library with the same name as the project
- A Dylan source file, called `module.dylan`, that defines a module with the same name as the project
- A Dylan source file for `.`. The file has the same name as the project, with `-info` appended, and with a `.DYLAN` suffix. So for a project `Hello` the file would be `hello-info.dylan`.

- An initial Dylan source file into which we can write application code. The file has the same name as the project, but with a `.DYLAN` extension.

### 4.1.1 Specifying the project name and location

The first page in the New Project wizard asks us to supply a name for our project, and to specify a location for the automatically generated project and source files.

When naming projects, you should note that Harlequin Dylan uses project names to generate the names of files, so you should only use characters that are valid in file names on your platform.

Since we want to develop a simple “Hello World” application, we are going to call the project `Hello`.

1. Type **Hello** in the Name box.

As you type **Hello**, the Location box fills in a folder for the automatically generated project and source files. We want to use this default value.

2. Type **Hello** in the Location box.

The New Project wizard prompts you if the Location folder does not exist. You must either create the default folder, or choose a different one before you can go on to the next page of the wizard.

We need not worry about the Advanced Project Settings section for now.

3. Click **Next**.

A dialog appears.

The dialog explains that the `Hello` directory does not exist, and asks you to confirm that you want it to be created.

4. Click **Yes** to create the `Hello` directory.

We now move to the second page.

### 4.1.2 Specifying the target type

The second page in the New Project wizard asks us to specify the kind of target we want to build from the project.

In addition, we can specify here that the wizard should generate template code for our project files. This template code provides a “skeleton” version of the sources for a project of the type we are creating. We can then modify the skeleton code to fit our needs. The contents of the skeleton code not only reflects the type of target selected on this page, but also answers to subsequent questions that the wizard asks.

In the Target Type box, the default option is “GUI Application (EXE)”. Our “Hello World” application is going to be an MS-DOS console application, so we need to change this option setting.

5. Select **Console Application (EXE)** in the Target Type box.

The library and module definitions in the new project will use whatever system libraries and modules are necessary for the target type we select. Thus an “OLE Compound Document Server” uses the DUIM-OLE-Server library.

6. Make sure the box labeled “Check this box to include any available templates” is checked.

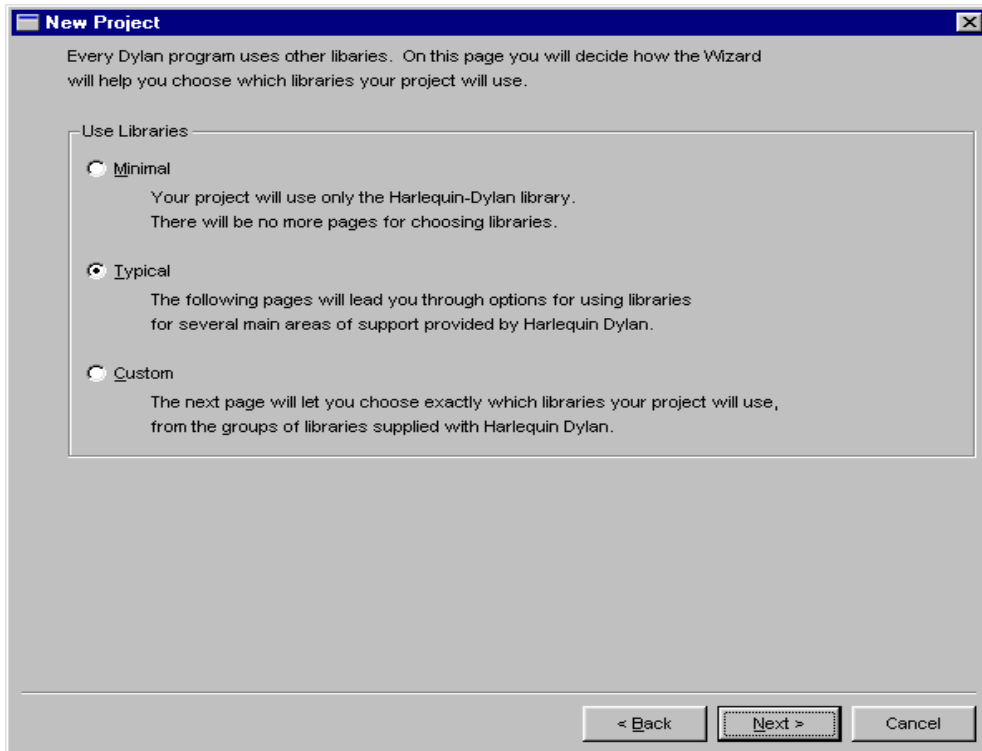


Figure 4.2 The New Project wizard's third page.

### 4.1.3 Choosing the libraries that the project uses

The next stage in creating our project is to decide which libraries and modules it is going to use. The third page of the wizard offers three different ways to do this. Each of the three options is described below.

#### 4.1.3.1 Minimal

If we choose the Minimal option, our project will use the Harlequin-Dylan library only.

Harlequin-Dylan is a convenience library that combines the standard Dylan library with a language extensions library called Harlequin-Extensions. Thus

Harlequin-Dylan provides a “Harlequin dialect” of Dylan. The standard Dylan library, without Harlequin’s extensions, is also available.

We need only Harlequin-Dylan for this project, so this would be the easiest option to choose. But we shall look at the other options first.

#### 4.1.3.2 Typical

If we choose the Typical option, the wizard takes us through a series of pages where we can select libraries by specifying our project requirements in high-level terms, without knowing the names of specific Harlequin Dylan libraries.

For instance, we can check the box for “Advanced file and directory handling”, and the wizard will set our project up to use the File-System library.

The Typical option is a useful way to create projects until you are more familiar with the libraries that Harlequin Dylan offers.

#### 4.1.3.3 Custom

If we choose the Custom option, the wizard allows us complete control over the libraries and modules our project will use. The wizard presents three list panes when we click **Next**. We can make library and module selections from each list pane.

At first, the only list available is the Choose Library Groups list. Because there are many libraries available in Harlequin Dylan, the wizard groups libraries by their functionality. We can select a group to see the list of libraries it contains, and then choose a library from the list. When we select a group, the wizard displays the library list in the second pane.

Libraries are grouped by functionality in a fairly broad fashion, so some libraries appear in more than one group because they fit more than one description. For instance, the Machine-Word library appears in both the **Core** group and the **Mathematics** group.

Notice that the Use column in Library Groups shows that **Core** is the only group from which a library or libraries will be used by default. If we select **Core**, we can see which libraries are going to be used.

1. Select **Core** in the Choose Library Groups list.

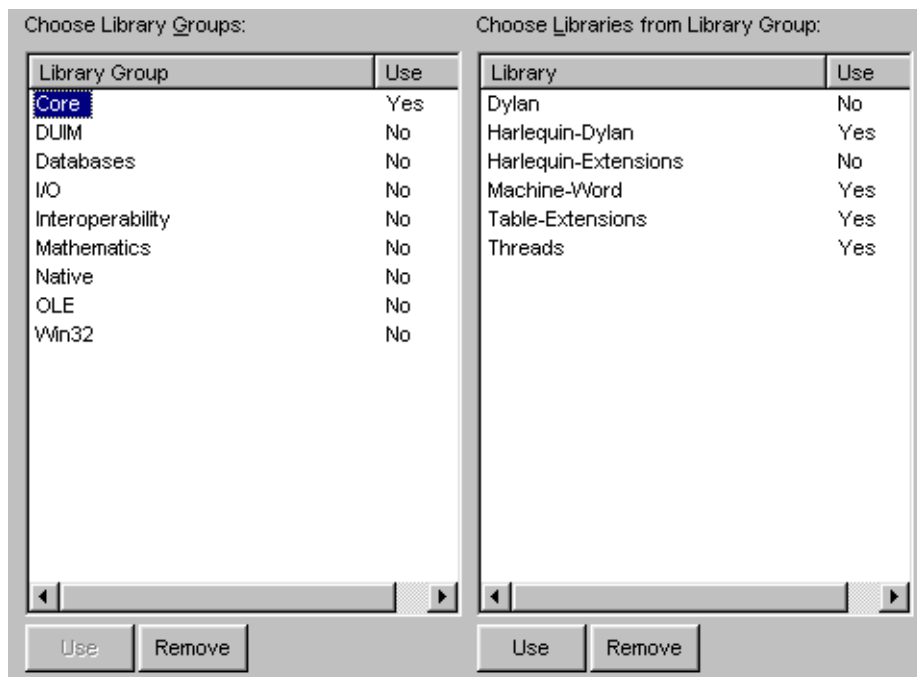


Figure 4.3 Harlequin-Dylan is the default library for use in new projects.

We can see that, by default, our project would use the libraries Harlequin-Dylan, Machine-Word, Table-Extensions, and Threads. We only need Harlequin-Dylan.

Harlequin-Dylan is a convenience library combining the standard Dylan library with a language extensions library called Harlequin-Extensions. This is what we want for our Hello project. Harlequin-Dylan has a module with a function that allows us to print our “Hello World” message, so we do not need to use any other libraries. That means we can remove the other libraries — Machine-Word, Table-Extensions, and Threads — by selecting them and clicking **Remove**.

If we now select Harlequin-Dylan in the Library list, we can see which modules from the Harlequin-Dylan library the project will include.

2. Select **Harlequin-Dylan** in the Library list.

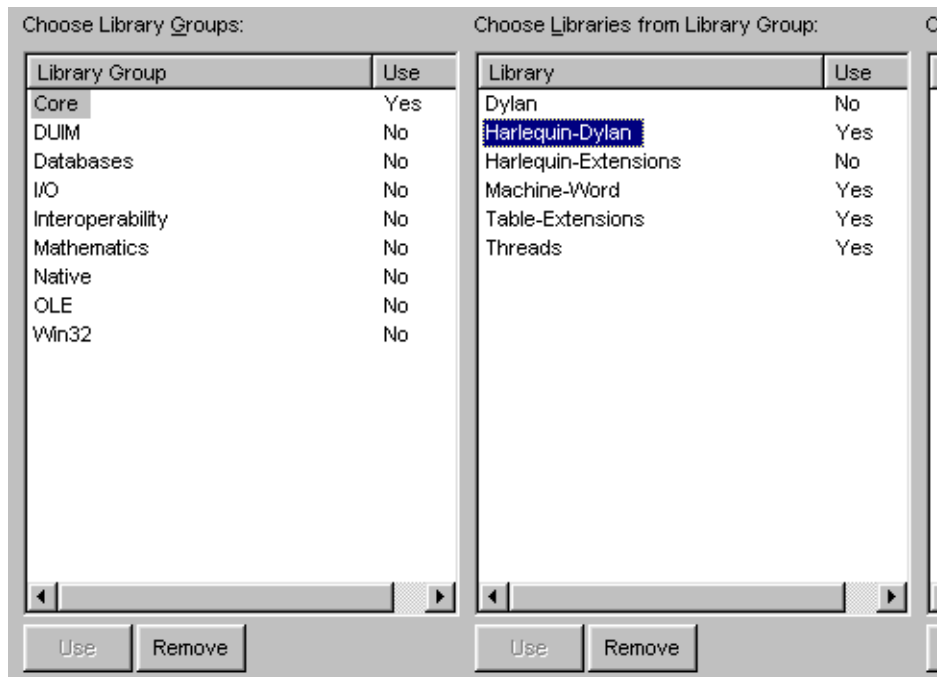


Figure 4.4 Default modules from Harlequin-Dylan for use in new projects.

These modules will suit our “Hello World” application fine, though we can remove simple-random from the list, as we have no need for random numbers, and we can also remove the finalization module, as we have no need to finalize any object instances. All we need to do is print a string to the standard output, and the simple-format module gives us a means of doing so. We must also keep harlequin-dylan.

Remember that, in Dylan, the library is the unit of compilation, and modules are simply interfaces to functionality within a library. By deciding not to use a particular exported module, you will import fewer interfaces into your application, but the application itself will not be any more compact. So using or not using simple-random and finalization will not make a difference to the size of the final, delivered application executable.

We now move to the third wizard page.

### 3. Click **Next**.



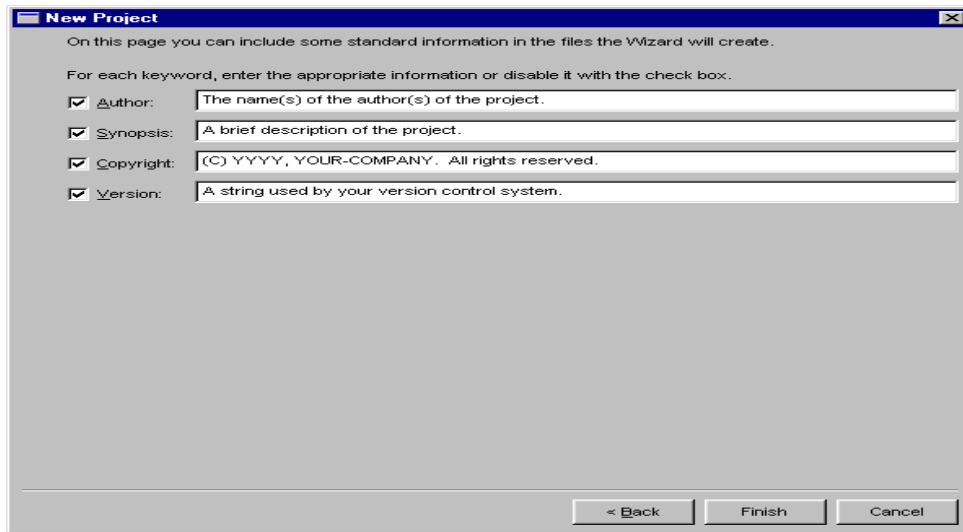


Figure 4.5 The New Project wizard's third page.

#### 4.1.4 The third page in the New Project wizard

The third, and final, page of the wizard gives us the option of supplying text for the documentation keywords Synopsis:, Author:, Copyright:, and Version:.

If we supply values for these keywords, the wizard will add them to the top of each of the files that it creates for the project, including the project file itself. With the exception of Synopsis:, these keywords are defined as part of the Dylan interchange format, on page 23 of the DRM. Synopsis: is a not a standard Dylan interchange keyword, but an additional one that Harlequin Dylan accepts.

1. Change the default keyword text as you see fit, or turn keywords off altogether.
2. Click **Finish**.

Now we have supplied all the information the wizard asks for, it creates the new project and opens it in the project window.

### 4.1.5 Examining the files in the Hello project

Figure 4.6 shows the Hello project opened in the project window.

The default view shows the Sources page, where we can see the three files `library.dylan`, `module.dylan`, and `hello.dylan`.

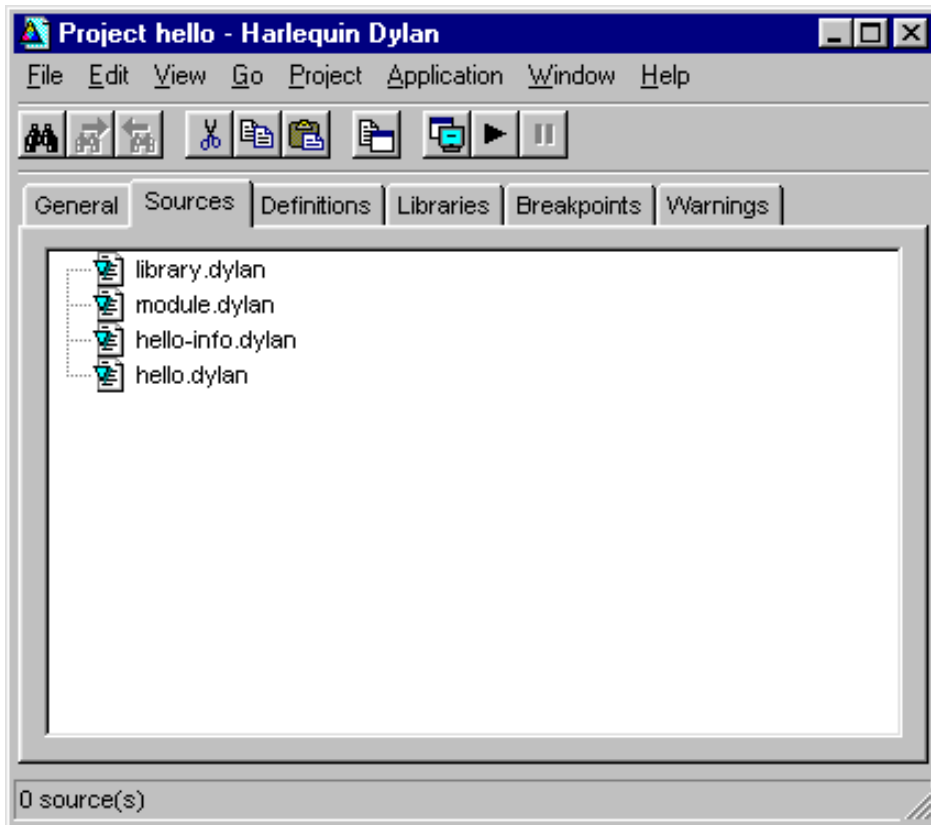


Figure 4.6 The Hello project.

The `library.dylan` file defines a Dylan library called `hello`, which uses the library `Harlequin-Dylan`. The `module.dylan` file defines a module of the `hello` library which is also called `hello`, and which uses various modules exported from the `Harlequin-Dylan` library; which modules these are

depends on whether you removed any from the list of defaults in the New Project wizard.

The `hello-info.dylan` file is .

The `hello.dylan` file is an initial file into which we can write the code for our project. It contains a default start function called `main`, and the last lines of the file call this `main` function.

We can add further files to the project as we see fit. But our “Hello World” application is trivial: we can write the code into `hello.dylan` now, and our work will be done. The application will simply call the function `format-out` on the string `"Hello World\n"`. The `format-out` function (exported from the `simple-format` module) formats its argument on the standard output.

1. Open the `hello.dylan` file in the editor.
2. Add the following code in the definition of `main`:

```
format-out("Hello World\n");
```

3. Choose **File > Save** to save the change to `hello.dylan`.

Now we can build our “Hello World” application.

4. Choose **Project > Build**.
5. Test the application by choosing **Project > Start**.

An MS-DOS console window appears, into which “Hello World” is written, and then the window exits.

The `hello.exe` can be found in the `bin` subfolder of the `hello` project sources folder we specified on the first page of the New Project wizard. See also “Projects on disk” on page 40 for more details.

## 4.2 New Project wizard persistence

When you click **Finish** on the last page of the New Project wizard, the wizard stores some of the choices and text-field settings you made so that they are available next time you create a project.

## 4.3 Advanced project settings

The **Advanced...** button on the first page of the New Project wizard leads to the Advanced Project Settings dialog.

The Library and Module names section allows you to specify names for your project's main library and module. The default value in both cases is the name of the project.

The Main function section allows you to change the name of the start function used in the project. See the **Project > Settings** dialog's "Debug page" on page 61 and "Examining the files in the Hello project" on page 58 for more details about this function.

The Version information section allows you to specify the major and minor version number of the project. You can also change these values after creating the project by going to the **Project > Settings** dialog.

The Compilation mode section allows you specify the compilation mode for the project. You can also change this setting after creating the project by going to the **Project > Settings** dialog.

## 4.4 Inserting new files and subprojects into a project

To insert a new file or subproject into a project, choose **Project > Insert File ...** in the project window. The project window prompts you with the **Insert File into Project** dialog, through which you can find the .DYLAN or .HDP file.

When you have chosen your file the project window places the file at the end of the existing list of project sources.

If you have added a subproject, remember that you will still need to edit the library and module definitions in your project to import from the new subproject.

## 4.5 Moving files within a project

To move a file to a new position in a project, select the file in the Sources page and use **Project > Move File Up** and **Project > Move File Down**.

## 4.6 Deleting files from a project

To delete a file from a project, select the file in the Sources page and choose **Project > Remove File**. You could also select **Edit > Cut**, **Edit > Delete**, or the “scissors” toolbar icon.

Harlequin Dylan asks you if you are sure you want to delete the file from the project, because you cannot undo the operation. Note that the file is not deleted from disk, just removed from the Sources list in the project. You can always put it back with **Project > Insert File**.

**Note:** Remember that the Definitions page will not be updated to reflect any definitions changes that deleting the file might imply, until you have rebuilt the project and caused the compiler database to be updated.

## 4.7 Project settings

The **Project > Settings ...** dialog allows you to set options for compiling and linking projects. There is a Compile page for compilation options, a Link page for linking options, and a Debug page for debugging options.

### 4.7.1 Compile page

Any project can be compiled in one of two modes. The Compile page controls the mode setting for the current project. See “Compilation modes” on page 44 for details of the modes.

### 4.7.2 Link page

The Link page controls whether a project is linked as an executable or as a DLL, and allows you to specify version information for the target. You can also change the name of the target here. The default name is derived from the name of the project.

### 4.7.3 Debug page

The Debug page allows you to specify a command line with which to execute the project target. This is especially useful for testing console applications from within the development environment. If there are values in the Com-

mand Line section of this dialog when you run a project target with **Project > Start** (and similar commands), Harlequin Dylan takes the contents of the Executable field, prepends them to the contents of the Arguments field, and sends the resulting command-line string to an MS-DOS console window for execution. Thus the values in these fields should form a valid MS-DOS command line.

The Debug page also allows you to change the start function for the project.

Recall that the Dylan language does not require an explicit start function for a project, such as `main` in C or Java. However, Harlequin Dylan allows you to record the name of a start function for a project. The debugger uses this name to know where to pause an application that you start up in interaction mode. See .

By default, this start function name is `main`, corresponding to the `main` function that the New Project creates by default in the *project-name.dylan* file of all new projects.

## 4.8 Project files and LID files

Harlequin Dylan's project files can be exported in a portable library interface format called LID (library interchange description). Harlequin and other Dylan vendors have chosen LID as the standard interchange format for Dylan libraries. LID files describe libraries in a flat ASCII text format for ease of interchange between different Dylan systems. The *Core Features and Mathematics* reference volume describes LID. LID files must have the extension `.lid`.

### 4.8.1 Importing a LID file as a project file

When you open a LID file in the development environment, it is converted into a project file and opened in a project window. (This process does not modify the original LID file on disk.)

In order to open a LID file as a text file in an editor, open the LID file using **File > Open** and select the file type filter **Dylan Library Interchange Descriptions (as text)** before clicking **Open**.

### 4.8.2 Exporting a project file as a LID file

To export a project file as a LID file for use in other Dylan implementations, use **File > Save As** and choose the file type **Dylan Library Interchange Descriptions**. Note that a LID file created by export will not contain any special build information, and the source files will be given with names only, and not locations.





# 5

---

---

## Learning More About an Application

**NOTE: THIS CHAPTER HAS NOT BEEN UPDATED SINCE HARLEQUIN DYLAN BETA ONE.**

In this chapter, we examine the Harlequin Dylan browser, using the code of the Reversi example.

### 5.1 About the browser

The browser can display information about every kind of Dylan object. With the browser, we can examine subclass-superclass relationships between class definitions; look at the arguments and return values of a generic function, generic function method, or macro; discover the generic functions and methods defined on a particular class; examine the state of runtime objects in a paused application thread; see the current state of application threads; and more. These features make the browser a powerful complement to the debugger for development work.

The browser gets its information by consulting a *browser database* for the current project. A browser database is a collection of information about a project that is derived when the project is compiled.

## 5.2 Browsing Reversi

To illustrate the features of the browser, we shall browse the Reversi application's code.

1. If it is not already open, open the Reversi project and start the Reversi application.

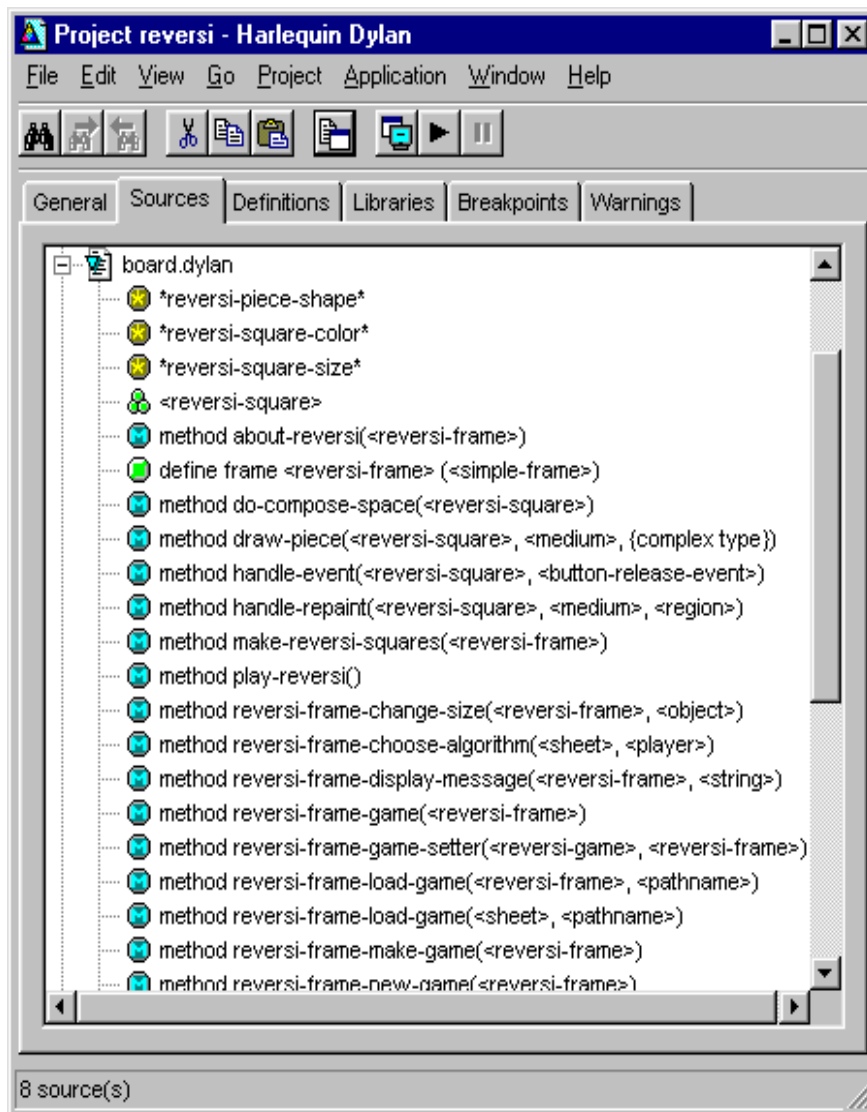


Figure 5.1 The definitions in the Reversi project.

2. Select the project tool's Definitions page.
3. Expand "library reversi", then "module reversi".

4. Double-click on `<reversi-frame>` in the alphabetically sorted list of definitions.

The browser appears.

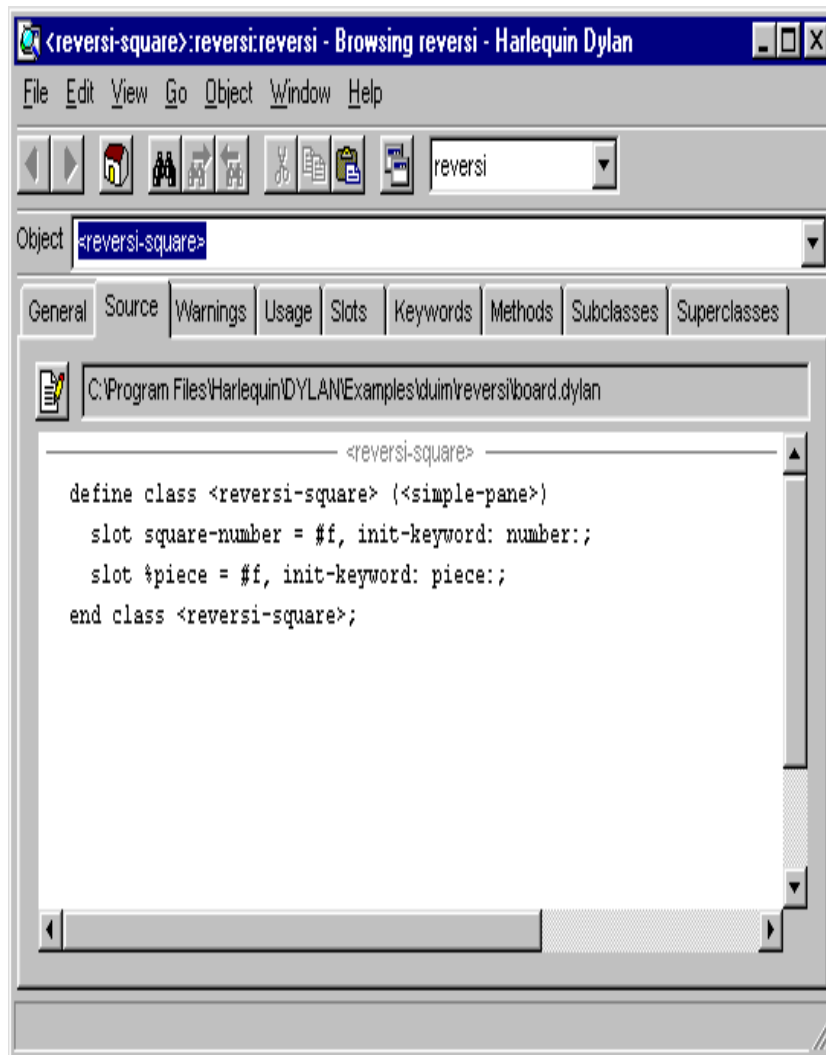


Figure 5.2 The Harlequin Dylan browser.

The browser shows the Source page for the class `<reversi-frame>`. This page shows the class's definition in `board.dylan`. (The browser gets the definition's source location from the browser database.)

When browsed, every Dylan object that has a definition will have a Source page.

All browsable objects have a General page, which contains information such as the name of the object, its type, its source location (if any), and so on. The other pages available in the browser depend on the kind of object being browsed. Table 5.1, page 70 shows the browser pages available for each major Dylan object type.

The General page for `<reversi-frame>` shows that it is a class from the source file `board.dylan`; that it is exported from the `reversi` library's `reversi` module; and that it has thirteen slots.

The Address field is only relevant when we are browsing an object in an application running under the debugger. We shall see more about that in Chapter 6, "Debugging and Interactive Development".

The drop-down list in the toolbar shows the module that the item being browsed belongs to. This modules list contains all modules exported by the object's application, as well as `dylan-user` and "none". Thus from the Object list and this list, we can tell that `<reversi-frame>` is defined in the module `reversi`.

If we change the selection in the modules list to a module that does not define `<reversi-frame>`, the presentation of the name in the Object list will change to `<reversi-frame>:reversi`. This is a shorthand way of stating that the name is defined in a different module to that selected in the modules list.

So the rules of name presentation in the Object list are:

1. If the name is visible in the current module (imported from a used module) it is not qualified.
2. If the name is not visible in the current module, it is presented with the name of the module in which it is defined as a suffix. If the name is defined in a used library (any library other than the library defined in the project), the library name is also added to the suffix.

What else can we find out about `<reversi-frame>`? The Errors page lists compiler warnings and errors generated the last time the class was compiled. The remaining pages — Usage, Slots, Keywords, Methods, Subclasses, and Superclasses — give the following information:

Usage	Used definitions and client definitions
Slots	Direct and inherited slots
Keywords	Direct and inherited initialization keywords
Methods	Direct methods
Subclasses	All subclasses
Superclasses	All superclasses

Table 5.1 Dylan object types and their browser pages.

Type	Browser pages available
Class	General, Source, Errors, Usage, Slots, Keywords, Methods, Subclasses, Superclasses
Generic function	General, Source, Errors, Usage, Methods
Method	General, Source, Errors, Usage
Macro	General, Source, Errors, Usage
Variable	General, Source, Errors, Usage
Constant	General, Source, Errors, Usage

## 5.3 Navigation with the browser

Navigation with the browser works like a World Wide Web browser does. When we are browsing an object, any other Dylan objects that are part of the current browser page are linked, so that we can browse one of those objects next if we want to. For instance, if we are browsing the subclasses of `<object>`, we can double-click on a subclass we are interested in, and the browser will take us to a view of that subclass. Successively browsed objects

appear in the same window, and we can navigate the browser object history using the arrowhead buttons on the browser toolbar.

1. While browsing `<reversi-frame>`, go to the Superclasses page.

The Superclasses page displays a class's superclasses using a tree view.

2. Expand the `<simple-frame>` item.
3. Double-click on the `<basic-frame>` item.

The browser switches to browsing the class `<basic-frame>`.

Notice that the backwards arrowhead button on the toolbar has become active. These arrow buttons allow you to navigate through the browser history.

4. Click on the backwards arrowhead button.

The browser returns to browsing the Superclasses page for `<simple-frame>`.





# 6

---

## Debugging and Interactive Development

**NOTE: THIS CHAPTER HAS NOT BEEN UPDATED SINCE HARLEQUIN DYLAN BETA ONE.**

In this chapter, we look at Harlequin Dylan’s debugger tool.

### 6.1 The debugger

The debugger is a tool for browsing and interacting with application threads. You can consider it a specialized version of the browser for browsing paused threads. Every thread in an application can be connected to its own, separate debugger.

The debugger provides the usual features you expect of a debugger — stepping, breakpoints, tracing, and so on — as well as a graphical interface for browsing the stack of the application thread to which it is connected. You can also use the debugger to interact with a paused application thread. Simply enter Dylan code at a prompt, and the code is executed in the context of the paused thread.

### 6.2 About the debugger

We now take a look at the basic debugger window panes.

The easiest way to bring up the debugger is to choose **Application > Debug** in the project tool, which debugs the main application thread. If you are already running Reversi, you must stop it first with **Application > Stop**.

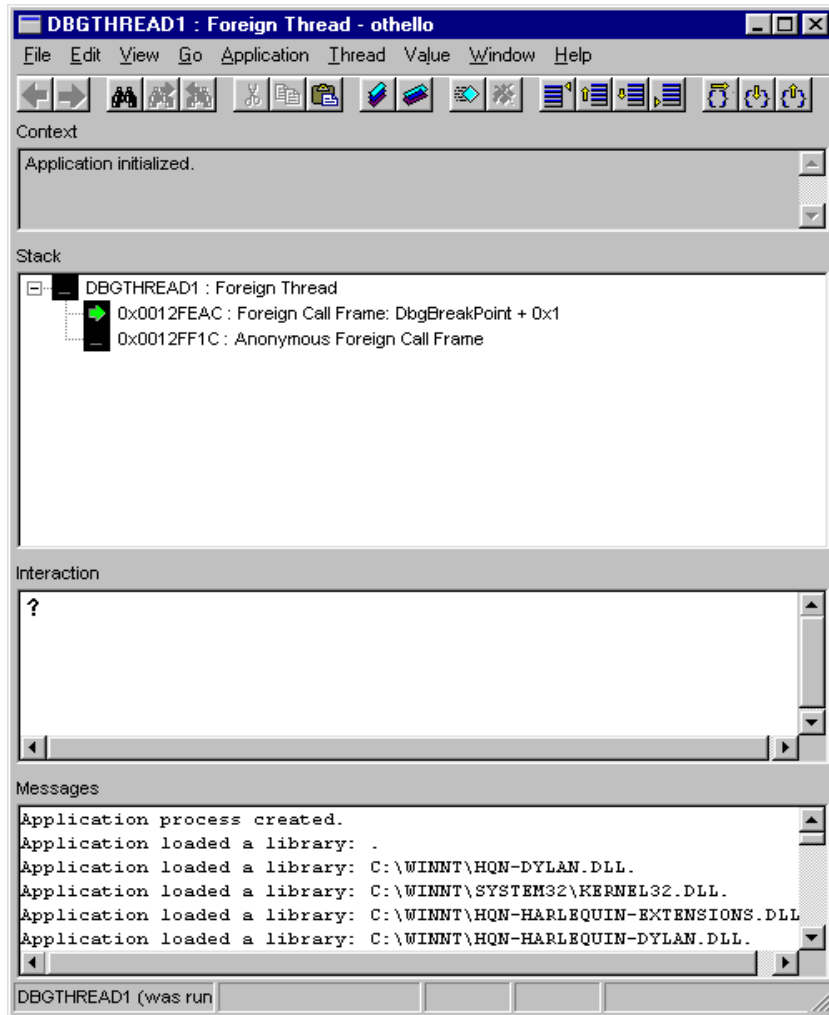


Figure 6.1 The debugger.

The debugger has four panes: the context pane, the stack pane, the interaction pane, and the messages pane.

**Note:** Because it can be costly to update the stack pane for every debugger window, Harlequin Dylan only updates the pane when you ask it to with **View > Refresh**. If you choose **View > Refresh All Debuggers**, Harlequin Dylan will update every debugger window currently open.

### 6.2.1 Context pane

The pane at the top of the debugger is the *context* pane. The context pane gives an overview of the state of the application to which the debugger is attached.

At any time, the context pane will either be blank or will display a message. A blank pane indicates either that the application is running, or that you paused the application by choosing **Application > Pause**, or that some other thread encountered a run-time exception.



Figure 6.2 The debugger's context pane.

A context pane message explains why this particular debugger's thread paused the application. Often the message will be the description of an unhandled Dylan error, but it could describe an out-of-language error (for example in foreign code) or one of a number of application events on which you can ask the debugger to pause the application, such when a library is loaded. You can see a list of possible exceptions and the actions that will be taken upon them in the Debugger Options dialog. See Section 6.9 on page 84.

### 6.2.2 Stack pane

The *stack* pane shows the thread's control stack at the moment it was paused. It depicts the stack in a tree view.

At the highest level, the stack pane lists the frames of the control stack. Each function call frame is represented by the name of the function whose call created the frame. (Tail calls create a stack frame.)

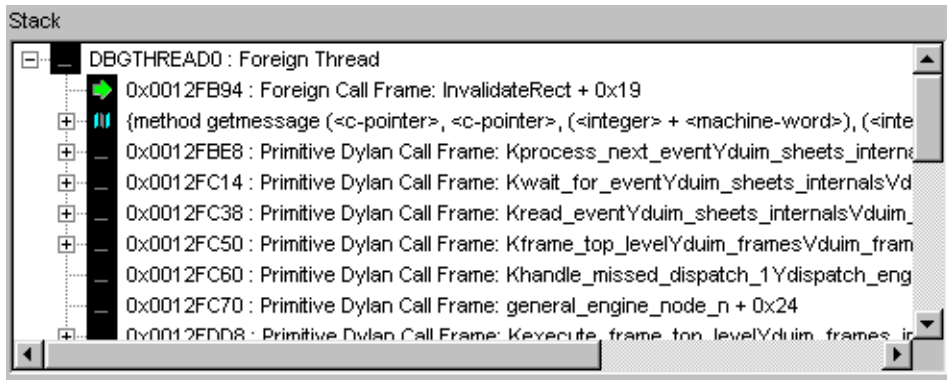


Figure 6.3 The debugger's stack pane.

You can also see local variables in the backtrace pane, by expanding the tree view. The menu commands **View > Expand**, **View > Expand All**, **View > Collapse**, and **View > Collapse All** control expansion in the tree view.

### 6.2.3 Interaction pane

The interaction pane allows you to enter Dylan expressions for evaluation in the context of the paused thread.

The **Application > Interact** command allows you to start an application up to interact with its main thread. You can also interact with any paused thread you are currently debugging.

To interact with a paused thread, simply type valid Dylan expressions at the prompt and they will be compiled and executed in the context of the current thread. Thus you can query and set values in an application, and then resume executing it.

Output in the interaction pane is “live”. If you select some output and click the right mouse button, you will see a menu of operations possible on that output.

### 6.2.4 Messages pane

The messages pane collects messages sent from the application and generated by the debugger. These are:

- debug messages
- error messages
- warnings
- other exceptions (for example, library loading and unloading)

The pane does not collect ordinary output from the application being debugged. Such output continues to go to its ordinary destination.

## 6.3 Debugging a specific application thread

As we noted earlier, each application thread can have its own debugger. The command we have seen so far, **Application > Debug**, debugs only the application's main thread. To debug another thread in the application, you should select **Application > Threads...** from the project window or editor, after first pausing the application.

The command launches a browser on the application itself, treating it as an object consisting of one or more threads whose states are visible in a table. If you browse a particular thread, Harlequin Dylan opens a debugger window on it. You can browse a thread in the table by using **Object > Browse**, or double clicking, or by using the right-click popup menu.

## 6.4 Controlling application execution

The **Application** menu, shared by the debugger, editor, and project windows, contains a set of commands for controlling the execution of an application. Some toolbar buttons provide shortcuts to these commands.

In a project window, the commands on the **Application** relate to the executable application that was last built for that project. (We have seen this in previous chapters.) In an editor, the commands relate to the executable application that was last built for the project of which the source file being edited forms a part. In the debugger, the commands relate to the application of which the thread being debugged forms a part.

### 6.4.1 Starting applications

**Application > Start** executes the application with which the window is associated.

This command is never available in debugger windows, because debugger windows are always associated with a thread in an application that has already started running.

After you have started executing an application, the **Application > Start** command is never available again until you stop the application (with **Application > Stop**) or until the application terminates normally.

### 6.4.2 Debugging applications

**Application > Debug** starts executing the application with which the window is associated, then pauses it as soon as the main thread initializes, and opens a debugger window on the main thread.

### 6.4.3 Stopping applications

**Application > Stop** terminates the application with which the window is associated. After you have stopped an application in this way, you can start it again with **Application > Start**.

**Note:** The command **Application > Restart** is a shortcut combining equivalent to **Application > Stop** followed by **Application > Start**.

### 6.4.4 Restarting applications

**Application > Restart** restarts the application with which the window is associated.

This command is only available if the application is already running. Since restarting an application logically consists of stopping it and starting it again, choosing this command is equivalent to choosing **Application > Stop**, then **Application > Start**.

### 6.4.5 Pausing and resuming execution of applications

**Application > Pause** pauses the execution of the application with which the window is associated. When an application is paused, you can browse and debug its threads or interact with it. Choose **Application > Resume** to resume execution.

## 6.5 Changing the debugger layout

Harlequin Dylan arranges the debugger window according to context. If you are chiefly debugging an application, the context pane will be largest; if you are chiefly interacting with an application, the interaction pane will be largest. You can change the default layout using **View > Maximize Interactor** and **View > Maximize Stack.**, which control the relative size of the interaction and stack panes. **View > Maximize Messages** also the

## 6.6 Setting breakpoints on application code

Harlequin Dylan allows you to set breakpoints on application code from the editor or browser. You can set breakpoints on Dylan code lines in a source file or on suitable objects in the browser: generic functions, methods, and functions.

### 6.6.1 Setting breakpoints on lines of code

To set a breakpoint on a line of source code, file, open the source file in the editor, and click on the leftmost column. This sets a *pausing breakpoint* on that line of code. The pausing breakpoint is denoted by a solid red circle. When you next run the application, any thread that executes that line of code will pause and a debugger will open on it.

If you click on the solid red circle it changes to a hollow red circle. The hollow red circle denotes a pausing breakpoint that has been disabled. Thus you can cycle through the enabled, disabled, and cleared states for pausing breakpoints.

You can click the right mouse button while your cursor is over the leftmost column to get a popup menu with several breakpoint commands.

**Run to Cursor** Sets a transitory breakpoint at the line the mouse pointer is on, then starts the application or resumes the application if it was paused. The transitory breakpoint is denoted by a solid green circle.

**Set Breakpoint** Sets an enabled pausing breakpoint at the line the mouse pointer is on.

**Clear Breakpoint**

Removes any breakpoint at the line the mouse pointer is on.

**Edit Breakpoint Options...**

Pops up a dialog for editing the breakpoint's options. It will create a breakpoint if there wasn't already one there. See Section 6.6.2 on page 80.

**[x] Breakpoint Enabled?**

A check item for toggling whether the breakpoint at the line the mouse is on is enabled or disabled.

## **6.6.2 Breakpoint options**

The Edit Breakpoint Options... dialog contains the following fields:

**Enabled[x]** Check box for toggling the whether a breakpoint is enabled or disabled. A disabled breakpoint does not affect the application's execution. By default a new breakpoint is enabled.

**Pause application [x]**

Check box for toggling whether the breakpoint pauses the application when it is encountered.

By default, a new breakpoint does pause the application.

**Print message [x]**



	Check box for toggling whether the breakpoint prints any message in the debugger's messages pane when it is encountered. By default a new breakpoint does print a message.
One shot [ ]	Check box for toggling whether the breakpoint is transient or permanent. Transient breakpoints are removed after they have been encountered. By default new breakpoints are permanent. Run to Cursor creates transient breakpoints.
Message Text	Text field for entering some identifying message to be associated with the breakpoint (if any). The text is used in messages referring to the breakpoint.

### 6.6.3 Breakpoint information in the project window

The project window has a Breakpoints tab that lists the current breakpoints. The same icons are used to identify the type of breakpoint as are used in the editor. The two other columns are:

Location	The place where the breakpoint is set. For breakpoints set at lines in the editor this will be a file name and a line number. For breakpoints set on functions this will be the name of the function.
Message	Any message text associated with the breakpoint.

The same right mouse button breakpoint menu is available on these items as is available in the editor.

### 6.6.4 Breakpoint information in the browser

When browsing an object that can have a breakpoint set on it, such as a function, generic function, or method, the popup menu available from the right mouse button menu includes the breakpoint manipulation popup menu as a component.

### 6.6.5 Breakpoint information in the debugger

When a breakpoint is encountered the breakpoint location is printed in the context window, and also (if the Print Message check box is enabled) added to the Messages window.

**Note:** Two distinct breakpoints may not have distinct underlying code locations, and so more than one breakpoint description could be printed.

## 6.7 Saving a bug report

You can save a formatted bug report for a thread by choosing **File > Save Bug Report** in the debugger for that thread. The command saves a bug report in a text file.

The command inserts a stack backtrace into the report automatically. The Save Bug Report dialog contains an editor pane into which you can type text that will be included at the top of the bug report. You can also insert a bug report template if you have one.

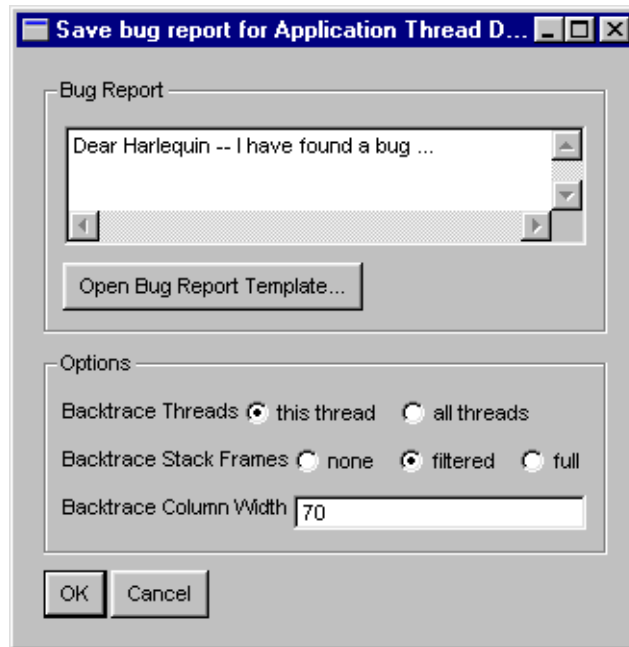


Figure 6.4 The Save Bug Report dialog.

You can modify the form of the stack backtrace that will be inserted into the bug report.

#### Threads

Choose one of “This thread” (the default) and “All threads” (includes every application thread in the backtrace).

#### Stack frames

Choose one of “None” (the backtrace will be empty), “Filtered” (excludes frames from the backtrace as specified on the **View > Options** dialog’s Stack page; this is the default), and “Full” (gives a complete, unfiltered backtrace).

### Column width

Enter an integer value here for the number of characters per line in a bug report. The default value is 70.

## 6.8 Refreshing debugger views

If a debugger for a particular application thread is already mapped on the screen, and a new event occurs in that thread which affects the validity of the information displayed in its debugger — such as the thread starting, stopping, or failing to handle an error — Harlequin Dylan updates the debugger display automatically.

However, Harlequin Dylan does not update any other debuggers that happen to be mapped for other threads. The **View > Refresh** command will update the information displayed in such a debugger; **View > Refresh All Debuggers** will update the information displayed in all debuggers currently mapped.

**Note:** If you pause the application with the Pause toolbar button, or by choosing **Application > Pause**, the pause event you caused is not actually an event within any application thread — it is an external event. In this case, no debugger window is refreshed; you must refresh the windows you want by selecting one of the explicit refresh commands on the **View** menu.

## 6.9 Debugger options

The **View > Options** command brings up a dialog that controls the options for the debugger. This dialog has three property pages: Stack, Exceptions, and Misc. The options on these pages apply on a per-thread basis — you can have different settings for different threads' debuggers.

### 6.9.1 Stack options

The Stack page controls how backtraces are displayed in the debugger's backtrace pane.

### Stack Frame Types

Check the boxes to include any of the following frame types: Dylan function calls", "Foreign function calls", "Cleanup frames", and "Unknown stack frame types".

### Stack Frame Libraries

Choose one of "Current library" (include frames whose corresponding definitions are defined in the current library only), "Used libraries" (include frames from the current library and the libraries it uses; the default), and "All libraries" (include frames from all libraries).

### Stack Frames Matching

Enter a string in the Include text box; frame names matching this string will be included in backtraces.

Enter a string in the Exclude text box; frame names matching this string will be excluded from backtraces.

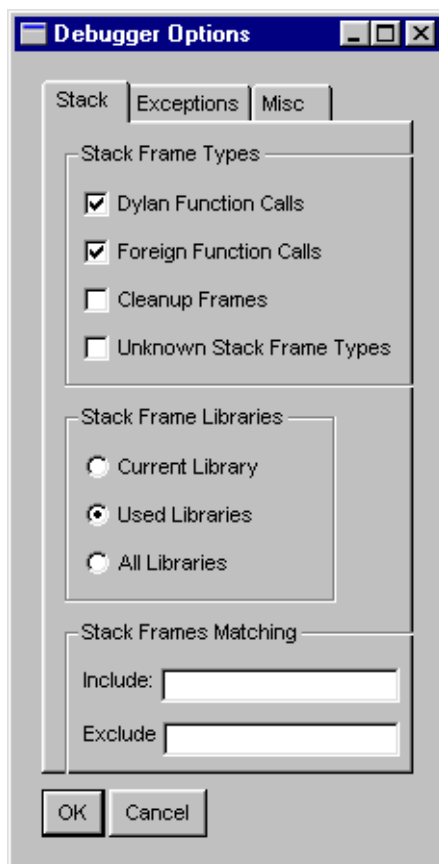


Figure 6.5 Debugger stack backtrace options.

### 6.9.2 Exceptions options

The Exceptions page controls the action taken when a particular exception occurs in the thread. Use the Action popup list to select an action.

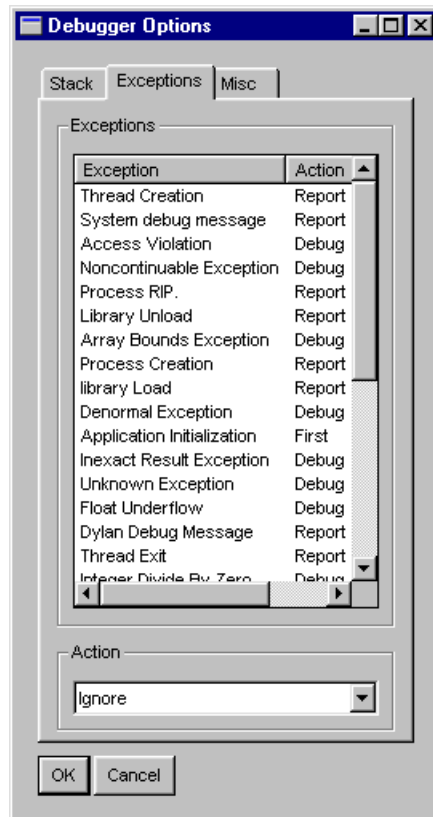


Figure 6.6 Debugger exceptions options.

The possible actions are:

- |        |  |
|--------|--|
| Ignore | Throw the exception away and allow the application to continue.  |
| Debug  | Stop the application. Update the debugger for the thread that signalled the exception. Write the message into a log. Activate any other debugger panels, but without updating them automatically. Allow an arbitrary amount of debugging and continue executing the application when <b>Application &gt; Resume</b> is selected. |

Report                      Write the message into a log and continue.

### **6.9.3 Miscellaneous options**

The Misc page presents miscellaneous debugger options.

When checked, the ‘Always debug’ option causes the debugger to pop up by default when it catches an unhandled exception. When the option is unchecked, you will be asked whether you want to debug.



---

---

# Dispatch Optimization Coloring in the Editor

This chapter is about optimization coloring and deals mainly with dispatch optimization details for now. Other optimization discussions forthcoming.

## 7.1 About dispatch optimizations

When you call a generic function in Dylan, the method that will be executed has to be selected from the set of methods defined on that generic function. The method is selected by comparing the types of the arguments passed in the generic function call to the parameter lists of the methods available; the method whose parameter list is closest to the types of the arguments passed in the call is the one that will be selected. Note that there may be situations where no one method is more applicable than another, or even where there is no applicable method at all — these situations may be detected at compile time or not until run time.

The process of selecting the right method to call is known as *method dispatch*. The algorithm for selecting a method is described in Chapter 6 of the DRM.

Method dispatch can, in principle, occur entirely at run time; but in some circumstances, the Harlequin Dylan compiler can work out at compile time precisely which method needs to be called, and so optimize away the need for run-time dispatch, making delivered applications faster. Depending on the circumstances, these *dispatch optimizations* can consist of inserting a direct call

to the correct method; working out that a particular generic function call amounts to no more than a class slot access and generating code to do that; or inlining the call completely.

## 7.2 Optimization coloring

When you compile a project, the Harlequin Dylan compiler records the kinds of optimizations it performed for each source code file in the project. It also records cases where compile-time optimization was for one reason or another not possible.

The Harlequin Dylan editor provides a way to see this information, by choosing **View > Color Dispatch Optimizations**. This command colors a source code file so that you can see where the compiler managed to optimize method dispatch, and also places where you may be able to make changes that will make dispatch optimizations possible next time you compile the project.

For instance, you will see a generic function call colored in blue where the compiler worked out the exact method to call and inserted a direct call to that method in the compiled code. Table 7.1 contains a full list of colors and their meanings.

### 7.2.1 Editing colored source code

Dispatch optimization coloring in a file shows only what the compiler achieved the last time you compiled the project containing the file. If you edit colored source code, the changes you make could affect the optimizations the compiler can carry out upon it the next time you compile the project.

For this reason, when **View > Color Dispatch Optimizations** is on and you edit a colored line, the editor resets that line only to black. All the other lines in the file keep their color. In this situation, you can re-color the line by doing **View > Refresh**. Note that once you have edited the text, the coloring information may no longer fit it, but you may still find it useful.

### 7.2.2 Effect of compilation mode on dispatch optimizations

The Harlequin Dylan compiler tries to optimize method dispatch whether compiling a project in Interactive Development mode or in Production mode. But because the compiler does more optimization in Production mode, you will see more coloring information after a Production-mode build.

### 7.2.3 Incremental compilation and dispatch optimization information

This section yet to be written.

### 7.2.4 Dispatch optimization colors and their meanings

The following table shows the colors used to indicate different kinds of dispatch optimization.

Color	Meaning	Recommended action
Magenta	Call not optimized because the compiler could not determine all the applicable methods.  Happens when the generic function is open.	Where possible, turn open protocols into sealed protocols.
Red	Call not optimized despite the compiler finding all the applicable methods.  Can happen when the type of an argument is not specific enough.	Where possible, add more type information to the call site — to local variables, for example.

**Table 7.1** Dispatch optimization color meanings.

Color	Meaning	Recommended action
Blue	Call optimized. The compiler found the appropriate method, and inserted a direct call to it.	None required.
Green	Call optimized. The compiler found that this call was an access to a slot whose position in a class is fixed. It replaced the function call with (faster) code to access that slot value.	None required.
Dark gray	Call optimized. The compiler inlined the code of the appropriate method.	None required.
Light gray	This code was eliminated completely.	None required. (Unless the code should have been called.)

Table 7.1 Dispatch optimization color meanings.

## 7.3 Optimizing the Reversi application

In this section we look at the dispatch optimization color information for part of the Reversi application and see what we can do to optimize it.

Before doing that, we should build the Reversi application in Production mode so we know that the application has been optimized as much as possible.

1. Open the Reversi project.
2. Choose **Project > Settings** and, on the Compile page, set the compilation mode to **Production mode**.
3. Choose **Project > Clean Build**.

4. When the build is complete, go to the Sources page and open the file `board.dylan`.

An editor window showing `board.dylan` appears.

5. In the editor window, turn on the **View > Color Dispatch Optimizations** check item.

We can now see color information showing how dispatch optimizations were or were not carried out during the last build.

6. Go to the definition of the method `update-reversi-frame`.

You can use **Edit > Find** or the “binoculars” toolbar button to do this.

This is the definition of `update-reversi-frame`:

```
define method update-reversi-frame
  (frame :: <reversi-frame>) => ()
  let game = reversi-frame-game(frame);
  let board = reversi-game-board(game);
  for (piece in reversi-board-squares(board),
      square in reversi-frame-squares(frame))
    unless (square-piece(square) = piece)
      square-piece(square) := piece
    end;
  end;
  force-display(port(frame));
end method update-reversi-frame
```

The coloring for this definition shows a number of different optimization results.

The calls to `reversi-game-board` and `reversi-board-squares` are colored green. This means the compiler determined that the calls were equivalent to slot accesses — particularly, accesses to slots having a fixed offset from the memory address at which their class is located. In each case, the compiler removed the need for run-time method dispatch by replacing the call with code to access the location that would contain the slot value.

The calls to `reversi-frame-game`, `reversi-frame-squares`, and `port` (from the DUIM library), are colored blue, signifying that the compiler managed to determine precisely which method to call, and inserted a direct call to that method in the compiled application. Again, the need for run-time method dispatch was averted.

The equality test `square-piece(square) = piece` is colored in magenta. Here, then, is a generic function call case that was not optimized. Magenta coloring means that for this call to `=`, the compiler could not determine the complete set of methods from which it could attempt to select the appropriate method to call.

In the remainder of this example, we will make changes to this method and to other methods that it calls in order to optimize this call to `=`. In doing so, we will have to make other optimizations: these optimizations will make our ultimate goal of optimizing this equality test possible, as well as causing optimizations in other parts of the Reversi application.

Our first change is to make the definition of the method for `square-piece` (near the top of `board.dylan`) to be sealed. The result of a call to `square-piece`, of course, is one argument in the equality test that we are trying to optimize.

7. Go to the definition of `square-piece` and add the `sealed` definition adjective.

After the change, the definition of `square-piece` should look like this:

```
define sealed method square-piece
  (sheet :: <reversi-square>) => (piece :: <piece>)
  sheet.%piece
end method square-piece
```

The change tells the compiler that the application is not permitted to define any methods on the `square-piece` generic function at run time. With this knowledge, the compiler knows it has access to the complete set of methods on this generic function, and therefore can attempt to do the method selection itself.

We can recompile the application to see what effect our change has had.

8. Rebuild the application, and refresh the color information for `board.dylan` with **View > Refresh**.

The refreshed coloring shows the `square-piece(square)` part of the equality test in red. This confirms that the compiler knows all the applicable methods, but for some reason could not work out which method was the right one to call.

The reason for this is that the compiler does not know the type of the index, `square`. If we constrain this to be of type `<reversi-square>` we will improve the compiler's chances of selecting the right method on `square-piece`. While we are doing this, we can constrain the type of `piece`, and see what effect that has, if any.

9. Go back to the definition of `update-reversi-frame` and constrain `square` in the `for` loop to be of type `<reversi-square>`, and `piece` to be of type `<piece>`.

After the changes, the `for` loop declaration should look like this:

```
for (piece :: <piece> in reversi-board-squares(board),
     square :: <reversi-square> in reversi-frame-squares(frame))
  ...
end;
```

10. Rebuild the application, and refresh the color information for `board.dylan` with **View > Refresh**.

The refreshed coloring information shows two new optimizations. First, the call to `square-piece(square)` in the equality test is colored blue. This means the compiler determined the right method to call, and the application now makes a direct call to that method instead of performing dispatch at run time.

Second, the assignment in the body of the `unless` term is now colored blue. The constraint we added on `piece` allowed the compiler to select the correct method on `square-piece-setter`.





# Appendix A

---

## The Stand-alone Compiler

**NOTE: THIS APPENDIX HAS NOT BEEN UPDATED SINCE THE FIRST HARLEQUIN DYLAN BETA.**

### A.1 Introduction

In Harlequin Dylan, you can compile Dylan code using the interactive development environment or the stand-alone compiler. This appendix describes the stand-alone compiler.

### A.2 About the stand-alone compiler

The Harlequin Dylan stand-alone compiler is an executable application called `dylan-compile.exe`. You can find it in the `bin` folder of your Harlequin Dylan installation.

The stand-alone compiler is primarily intended as a batch compiler for build scripting (such as for use in makefiles), but it also provides an interactive mode interface that you may find useful.

## A.3 Using the stand-alone compiler in batch mode

To use the Harlequin Dylan stand-alone compiler in batch mode, go to an MS-DOS prompt, and enter `dylan-compile`, followed by command options and a list of one or more projects to perform the commands upon. and (optionally) a list of projects to compile. The basic form of this call is:

```
dylan-compile options projects
```

The default behavior is to compile any changes to the project sources that have occurred since they were last compiled.

You can specify projects either with a pathname to a project file `hdp` or LID (`.lid`) file, or as a library name.

So:

```
dylan-compile c:\users\dylan\projects\my-project\my-project.hdp
```

will compile incrementally as necessary. Similarly

```
dylan-compile foo-works bar-user
```

will incrementally compile the two libraries, if they can be found. The compiler will exit in error if it cannot find them.

Option	Description
<code>/?</code>	Prints help on using <code>dylan-compile</code> .
<code>/help</code>	Prints help on using <code>dylan-compile</code> .
<code>/nologo</code>	Suppresses the Harlequin copyright banner.
<code>/all</code>	Recompiles the projects in their entirety, whether or not their sources have changed since they were last compiled.

**Table 7.1** Command-line compilation options.

Option	Description
/link	Links the projects into executables. Uses the projects' Link settings ( <b>View &gt; Options</b> ) to determine whether to link as EXEs or DLLs, and uses the Windows registry to find the default system linker.
/save	Saves the compilation results into databases for future use by environment tools.
/exe	Forces the projects to be linked as EXEs, ignoring their own Link settings (from <b>View &gt; Options</b> ).
/dll	Forces the projects to be linked as DLLs, ignoring their own Link settings (from <b>View &gt; Options</b> ).
/microsoft	Forces the projects to be linked using the Microsoft linker, ignoring the system linker setting in the Windows registry.
/gnu	Forces the projects to be linked using the GNU linker, ignoring the system linker setting in the Windows registry.

**Table 7.1** Command-line compilation options.

Examples:

1. Compile and link a library as an executable (EXE) file:  

```
dylan-compile /link my-executable
```
2. Compile a library, save a database for it, and link it with the GNU linker:  

```
dylan-compile /save /gnu my-dll
```
3. Recompile a project from scratch and link it as an executable.  

```
dylan-compile /all /exe c:/dylan/my-project.hdp
```

## A.4 Using the batch compiler interactively

The interactive mode of the batch compiler allows you to carry out several compilations over a period of time without having to restart the compiler each time. To start the batch compiler in interactive mode, enter `dylan-compile` without any arguments at an MS-DOS prompt.

```
MS-DOS> dylan-compile
```

```
Harlequin Dylan ...
Copyright (C) 1995, 1996, 1997, The Harlequin Group Limited.
All rights reserved.
```

```
dylan =>
```

You can find a list of available commands by entering `help` at the command line.

### A.4.1 COMPILE CHANGES command

Usage: `compile-changes project`

The stand-alone compiler's `compile-changes` command is equivalent to the environment's **Project > Compile Changes** menu command. It is also the default operation performed when running the compiler in batch mode. This operation performs an incremental compilation, that is it recompiles the specified project only enough to take into account any changes that have been made since it was last compiled. Thus simple changes to one line of a file should be much faster than if almost every file in a project has changed.

### A.4.2 COMPILE-ALL command

Usage: `compile-all project`

The stand-alone compiler's `compile-all` command is equivalent to the environment's **Project > Compile All** menu command, and is the same as running the compiler in batch mode with the `/all` option. It recompiles the specified project, and all the user projects it uses, from scratch. Typically, you only need this command when you need to guarantee a full compilation, such as when recompiling a project in order to release it.

### A.4.3 LINK command

Usage: `link {options} project`

The stand-alone compiler's `link` command is equivalent to the environment's **Project > Link** menu command. It does not compilation, but simply links whatever build products already exist for the project, and stores them in the project's `bin` subfolder.

The *options* are:

<code>/dll</code>	Link as a DLL. (Default is EXE).
<code>/force</code>	Force link the project and its subprojects (that is, the user projects it uses).
<code>/not-recursive</code>	Link only this project, not its subprojects (that is, the user projects it uses).
<code>/gnu</code>	Link using the GNU linker. (The default.)
<code>/microsoft</code>	Link using the Microsoft linker.

### A.4.4 BUILD command

Usage: `build {options} project`

The stand-alone compiler's `build` command is equivalent to using `compile-changes` and then `link` on the project. This command accepts all of the options for `compile-changes` and `link`. It is equivalent to the environment's **Project > Build** menu command.

### A.4.5 OPEN command

Usage: `open project`

Opens a project either by its library name or filename. Once a project is open, future compilations of any libraries that use the library defined by the project will use this open definition.

### A.4.6 PARSE command

Usage: `parse project`

Parses a project, producing an in-memory model of its code. The project can be specified either as a library name or as a filename. Equivalent to the environment's **Project > Parse** menu command.

### A.4.7 IMPORT command

Usage: `import project`

Creates a project file for a given Library Interchange Definition file. The project can be specified either as a library name or as a filename. Equivalent to the environment main window's **File > Import LID...** menu command.

### A.4.8 CLOSE command

Usage: `close project`

Removes all knowledge of a project from the compiler so that, for example, you can compile a different version of that project.

### A.4.9 CLOSE-ALL command

Usage: `close-all`

Closes all open projects. This is useful when you have finished with one set of projects and want to start compiling a completely different set.

### A.4.10 HELP command

Usage: `help`

`help command`

Displays interactive-mode compiler commands, all along with a brief description of how to use them.

If you specify a *command*, the compiler prints documentation on how to use that command.

### A.4.11 EXIT command

Usage: `exit`

Exits the compiler.

### A.4.12 QUIT command

Usage: `quit`

Exits the compiler. An alias of the `exit` command.

## A.5 Recovering from errors

### A.5.1 ABORT command

If the compiler crashes during an operation, use this command to abort that operation so that you can continue without having to restart the compiler completely.

If you wish to report the error, use the `debug` command while you are running the compiler inside a debugger. See Section A.6 on page 104.

### A.5.2 CONTINUE command

If the compiler crashes during an operation, use this command to choose between one of the available restarts by specifying the restart's number. Typically you will just want to abort the operation — use the `abort` command for this.

### A.5.3 DEBUG command

If the compiler crashes, this command will allow you to enter a debugger so that you can determine what is happening. Generally speaking, you should use the `batch-debug` application as your debugger, as this will produce a bug report that you can send to Harlequin. See Section A.6.

## A.6 Reporting bugs from the stand-alone compiler

Harlequin Dylan provides the application `batch-debug` in the top-level `bin` folder. You can use it to produce bug reports.

If you are running the stand-alone compiler in batch-mode, replace your usual call to `dylan-compile` —

```
dylan-compile [options] projects
```

with this —

```
batch-debug dylan-compile /debug [options] projects > compiler.log
```

If the you are running in interactive mode and the compiler crashes, you are offered the choice of either aborting the compilation or entering the debugger. To make sure that you are in a position to report potential bugs, run `dylan-compile` as follows:

```
batch-debug dylan-compile | tee compiler.log
```



# Appendix B

---

## Editor Options

**NOTE: THIS APPENDIX HAS NOT BEEN UPDATED SINCE THE FIRST HARLEQUIN DYLAN BETA.**

This appendix describes editor configuration options.

### B.1 Editor options

You can configure the Harlequin Dylan editor in many different ways. The editor configuration options are available from the Editor Options dialog, available by selecting **View > Options** in the editor.

### B.2 Editing styles

To make configuration simpler, the editor provides two editing configurations, “Windows” and “Emacs”. These editing styles configure the editor so that they behave similarly to either generic Windows editors or the GNU Emacs editor. The styles are simply particular combinations of the settings in the the Editor Options dialog, so you can modify them to produce variations that you find more convenient.

To set the editor up for Windows-style editing, go to the Editor Options dialog with **View > Options**, select the Restore page, and click **Restore Windows Defaults**. To set the editor up for Emacs-style editing, click **Restore Emacs Defaults** in the same dialog.

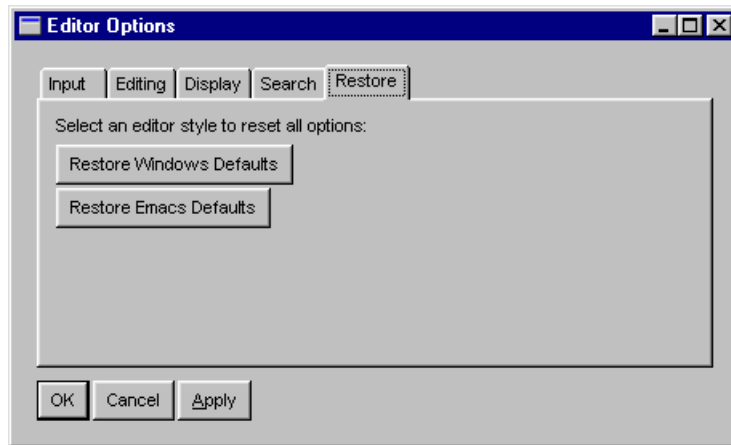


Figure 7.1 The Editor Options dialog's Restore page.

## B.3 The Editor Options dialog

### B.3.1 Editor input options

The Input page contains options for controlling how the editor handles keyboard and mouse input.

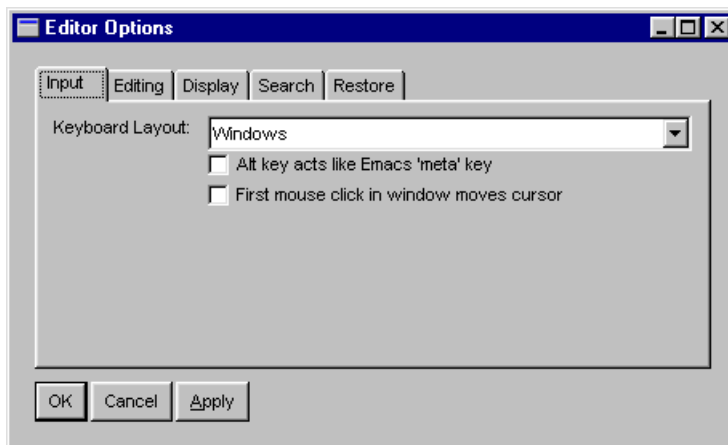


Figure 7.2 The Editor Options dialog’s Input page.

### Keyboard Layout

Sets Emacs-style or Windows-style key bindings. For example, in Windows mode, Ctrl+V pastes text, but in Emacs it moves the cursor down one page.

In Windows mode, the selection is “Windows” and in Emacs mode the selection is “Emacs”.

A full listing of Windows and Emacs key bindings is available in .

### Alt key acts like Emacs ‘meta’ key

If this box is checked, the Alt key acts like the Emacs Meta key in editor windows. For example, Alt+V moves the cursor up one page. If it is unchecked, the Alt key is used as normal for accessing menus; Alt+V would pull down the **View** menu.

In Emacs mode, this box is checked. In Windows mode, it is not checked.

### First mouse click in window moves cursor

If this box is checked, then first time you click inside the editor window the cursor moves to where you clicked. If it is unchecked, the cursor does not move.

This box is not checked in either Emacs mode or Windows mode.

## B.3.2 Editing

The Editor Options dialog's Editing page contains options related to manipulating files and their contents in the editor.

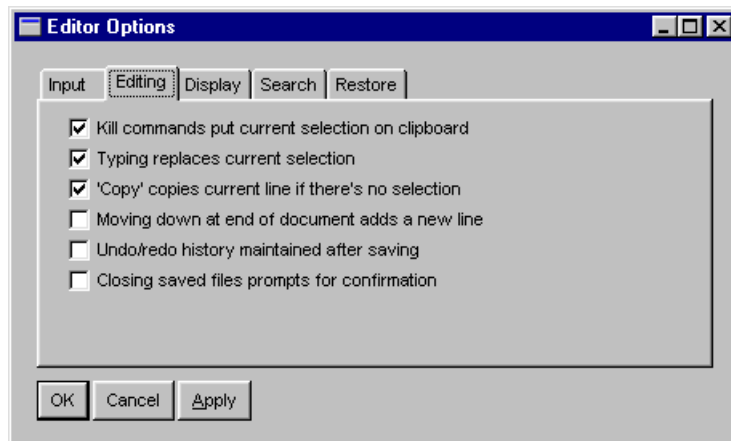


Figure 7.3 The Editor Options dialog's Editing page.

### Typing replaces current selection

If this box is checked, then typing a character when there is a selection deletes that selection and inserts the character you typed in its place. If this box is not checked, typing a character cancels the selection and then inserts the character as normal.

In Windows mode, this box is checked. In Emacs mode, it is not checked.

‘Copy’ copies current line if there’s no selection

If this box is checked, and you perform a copy operation when there is no selection, the editor copies the current line.

This box is checked in both Windows mode and Emacs mode.

Moving down at end of document adds a new line

If this box is checked, moving the cursor down at the end of a file inserts a new line. If it is not checked, the editor does not insert a new line and the cursor does not move.

In Windows mode, this box is not checked. In Emacs mode, it is checked.

Undo/redo history maintained after saving

The editor maintains a separate undo history for each file. If this box is checked, the undo history is retained after you save a file. If it is not checked, the undo history is emptied upon saving.

This box is not checked in Windows mode or Emacs mode.

Closing saved files prompts for confirmation

If this box is checked, the editor asks you for confirmation if you try to close a file that has been saved or has not been modified. If it is not checked, the editor closes such files without asking for confirmation.

In Windows mode, this box is not checked. In Emacs mode, this box is checked.

### **B.3.3 Display page**

The Editor Options dialog’s Display page contains options related to how the editor displays files and their contents.

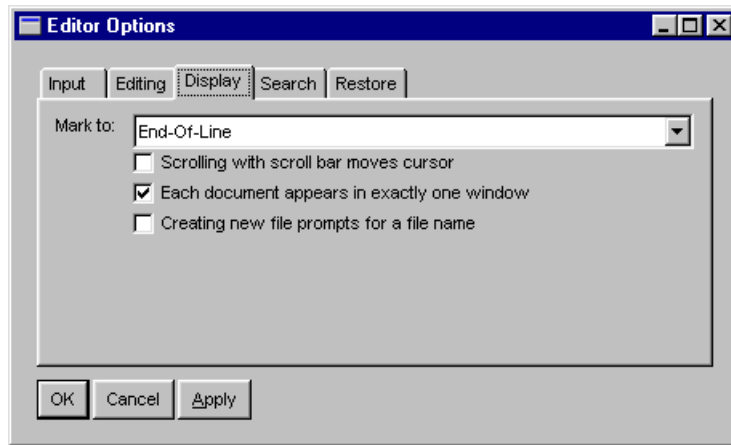


Figure 7.4 The Editor Options dialog's Display page.

**Mark to** Sets the position at which a selection to the end of a line ends. If you select *End-Of-Line*, the highlight extends only to the last character of each line. If you select *Right-Margin*, the highlight extends to the right-hand side of the window.

This option merely affects the way the highlight is displayed, and not the extent of the actual selection.

In Windows mode, the selection is *End-Of-Line*. In Emacs mode, it is *Right-Margin*.

**Scrolling with scroll bar moves cursor**

If this box is checked, and you use the scroll bar to scroll the window text, such that the cursor would move out of the visible area, the insertion point moves so that it remains in the visible area. If this box is not checked, the insertion point remains in its original place.

In Windows mode, this box is not checked. In Emacs mode, it is checked.

Each document appears in exactly one window

If this box is checked, each editor window shows exactly one file. Closing the window closes the file *and* closing the file closes the window. Before a file closes, you will always be prompted to save changes if appropriate.

If this box is not checked, there may be more files open than windows, and any window may show any file, using the **Go > File...** command, or the Emacs command C-x b.

In Windows mode this box is checked. In Emacs mode, it is not checked.

**Note:** It is possible for this box to be checked, but still have more than one window showing the same file. This can happen if you check this option after opening several such windows when it is unchecked. In that case, closing any of these windows, or closing the file in any window, closes all the duplicate windows.

Creating new file prompts for a file name

If this box is checked, the editor prompts you for a file name when you create a new file. If it is not checked, the editor assigns names to new files automatically.

In Windows mode, this box is not checked. In Emacs mode, it is checked.

### B.3.4 Search page

The Editor Options dialog's Search page contains options related to the editor's text searching facilities in the **Go** menu and the toolbar.

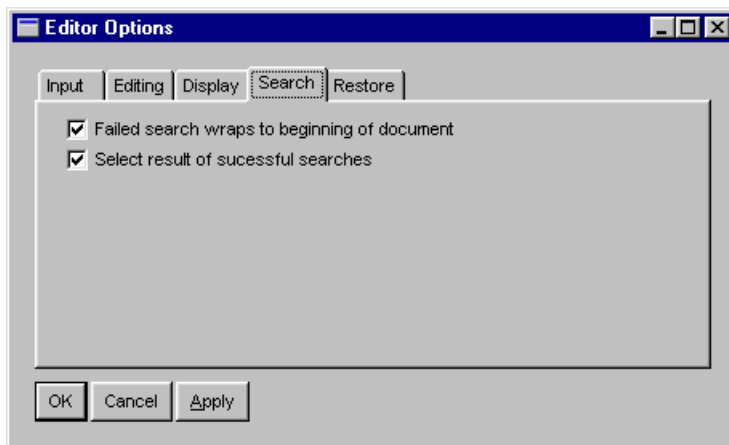


Figure 7.5 The Editor Options dialog's Search page.

#### Failed search wraps to beginning of document

If this box is checked, then if a search operation reaches the bottom of the file, the search “wraps around” and continues from the top of the file. If it is not checked, searches fail if they do not find a match before reaching the end of the file.

This box is checked in both Windows mode and Emacs mode.

#### Select result of successful searches

If this box is checked, then when a search succeeds, the editor selects the text it matched, just as if the text had been selected with the mouse or keyboard. If it is not checked, the editor does not select the matched text.

This box is checked in both Windows mode and Emacs mode.



### B.3.5 Restore page

The Editor Options dialog's Restore page restores the values for all the other pages to one of the two default modes, Windows or Emacs.

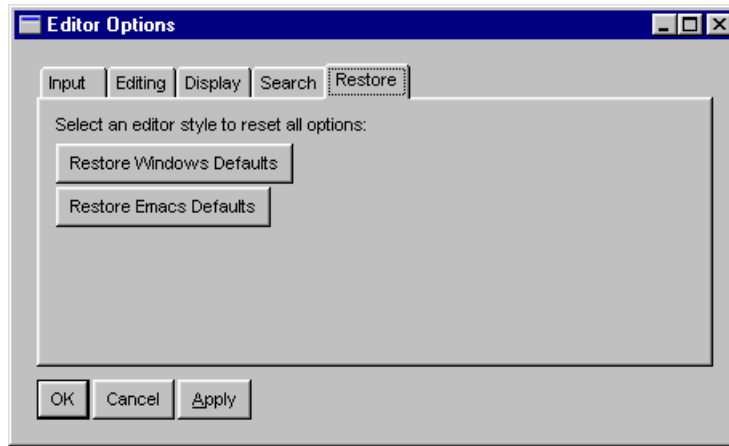


Figure 7.6 The Editor Options dialog's Restore page.

