

MORTEN NALLIN JENSEN
SMK-
OEDB-ELEMENT

**Rational Environment Training —
Fundamentals**

Copyright © 1986 by Rational

Document Control Number: 1010
Rev. 5, December 1985
Rev. 5.1, January 1986
Rev. 5.2, April 1986
Rev. 5.3, July 1986

This document subject to change without notice.

Rational
1501 Salado Drive
Mountain View, California 94043

Rational Environment Training — Fundamentals

Slides

Contents

Basic Mechanisms

Introduction	1
The Keyboard	4
The Screen	8
Environment Structure	14
Environment Traversal	18
Window Management	24
Command Execution	36
Help and Documentation	46
General Editing	51

Ada Program Creation

Basic Concepts	64
Ada Editing Aids	71
Ada Units	80
Unit Testing	91
Organization of Ada Units	96
More Ada Editing Aids	110
Multiple-Unit Ada Programs	116

Ada Program Modification

Simple Browsing	140
Introduction to the Debugger	165
Program Modification—Single-Unit Method	177

Program Modification—Multiple-Unit Method	204
Additional Topics	
Naming Conventions	213
Library Objects Management	221
Future Topics	239

Seminar Outline

Basic Mechanisms

- Introduction
- The Keyboard
- The Screen
- Environment Structure
- Environment Traversal
- Window Management
- Command Execution
- Help and Documentation
- General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Course Objectives

- To introduce the fundamental concepts and mechanisms of the Environment used in basic software development
- To provide experience in creating and changing small Ada programs using the Environment
- To build a foundation for further exploration in the Environment

Course Materials

- *Rational Environment Training—
Fundamentals*
 - Course slides
 - Hard copy of scripts
- *Rational Environment Basic Operations*
 - Sequence of steps, commands, and keys used to perform common Environment functions
 - Keymap
- *Rational Environment Reference Summary*
 - Keymap
 - List of Environment commands
- *Rational Environment Reference Manual*
 - 5-volume set

Seminar Outline

Basic Mechanisms

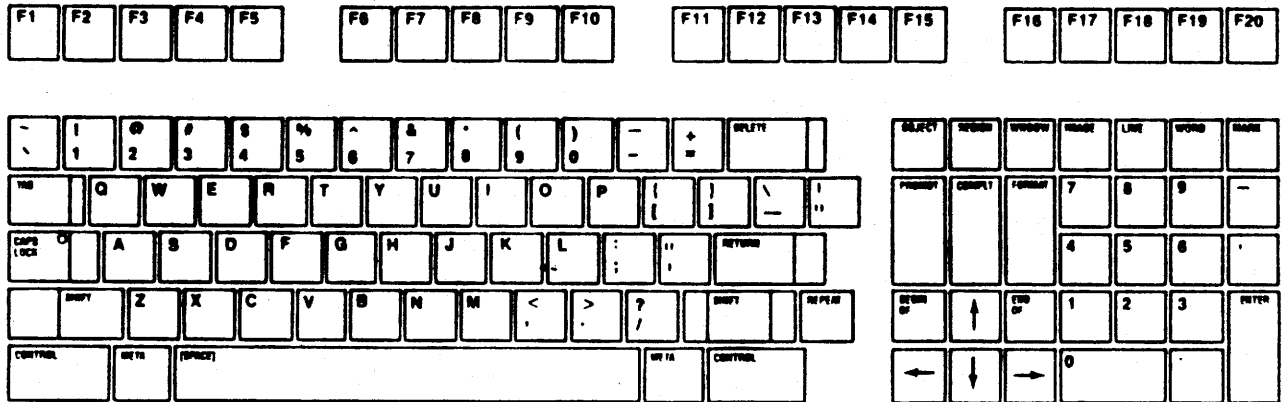
- Introduction
- The Keyboard
- The Screen
- Environment Structure
- Environment Traversal
- Window Management
- Command Execution
- Help and Documentation
- General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

The Keyboard *RATIONAL TERMINAL*



Key Usage

- Item - operation keys

- Items: `Object`, `Region`, `Window`, `Image`, `Line`, `Word`,
`Mark`

- Operations: for example, `↑`, `↓`, `Delete`,
`Begin Of`

- Press and release the item key, and then press and release the operation key

- Notation: `item key` - `operation key`

- Modifier keys

- Modifiers: `Shift`, `Control`, `ESCMeta`

- Operate with the function keys and basic alphanumeric keys

- Press and hold the modifier key while pressing the next key

- Notation: `modifier key` `Fn` OR `modifier key` `alphanumeric key`
`ESC` - `Fn`

Function Key Template

					CONTROL META SHIFT META SHIFT	Debug	Debug	Debug	Misc.	Traverse
					CONTROL META SHIFT	Debugger Window	Debug Show (Breaks)	Debug Task Display		Home
					CONTROL META SHIFT	Debug Stop	Debug Activate	Debug Propagate		Enclosing Object
					CONTROL SHIFT	Debug Run (Local) Debug Run	Debug Break Debug Definition	Debug Catch Debug Put	Print	Ada Other Part
						Debug Execute	Debug Display	Debug Stack	Prompt For	Definition
F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	
Help	List	Promote	Demote	Create	CONTROL META SHIFT META SHIFT	Errors	Items	Items	Jobs	Info.
Help Window	Library Space	Compilation Make	Compilation Demote	Create Text Create World	CONTROL META SHIFT				End of Input	What Object
Help on Key Help on Command	Verbose List List Objects File List	Code	Withdraw Source Demote	Create Directory Create Private Part Create Body Part Create Command	META CONTROL SHIFT	Clear Errors Show Errors	Item Off	Previous Item	Job Connect Job Kill Job Enable Job Disable	What Locks What: Users What Load What Time
Help on Help	Ada List	Install	Edit			Semanticize	Explain Item	Next Item		
F11	F12	F13	F14	F15	F16	F17	F18	F19	F20	

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

- The Screen

Environment Structure

Environment Traversal

Window Management

Command Execution

Help and Documentation

General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Login

- Login procedure

Commence Login

enter terminal type (RATIONAL):

enter user name: *PT-1*

enter password:

enter session name: *default S-1* *PT-1*

- Terminate each step by pressing Return

- Input is case insensitive

- Terminal types

— Valid terminal types: Rational, Vt100

— = at Commence Login OR enter user name prompts brings up the enter terminal type prompt *This error and terminal type should trigger*

— Return for the default type indicated in parentheses

Exercise: Logging In

Log into the Environment.

Types of Windows

- Message window: Displays system status information
- Major window: Displays images of objects (libraries, Ada units, files)
- Command window: Displays commands for execution
- Example

MESSAGE WINDOW

Rational Environment
 A_5_30_0 Copyright 1984, 1985, 1986, by Rational.
A COMMAND IS RUNNING

= Logo PT_1 S_1 *running*

MAJOR WINDOW

```

!!Users Pt_1 : Library (World):
Baseball_System      : Library (World);
Copyright_1986_Rational : File;
Debugging            : Library (Directory);
Experiment            : Library (World);
Project_1            : Library (World);
Rational_Commands   : Ada (Proc_Spec);
Rational_Commands   : Ada (Proc_Body);
Sample_File          : File;
Statistics_System    : Library (World);
S_1                  : Session;
S_1_Lost_Keys        : Pipe;
S_1_Switches         : File (Switch);
  
```

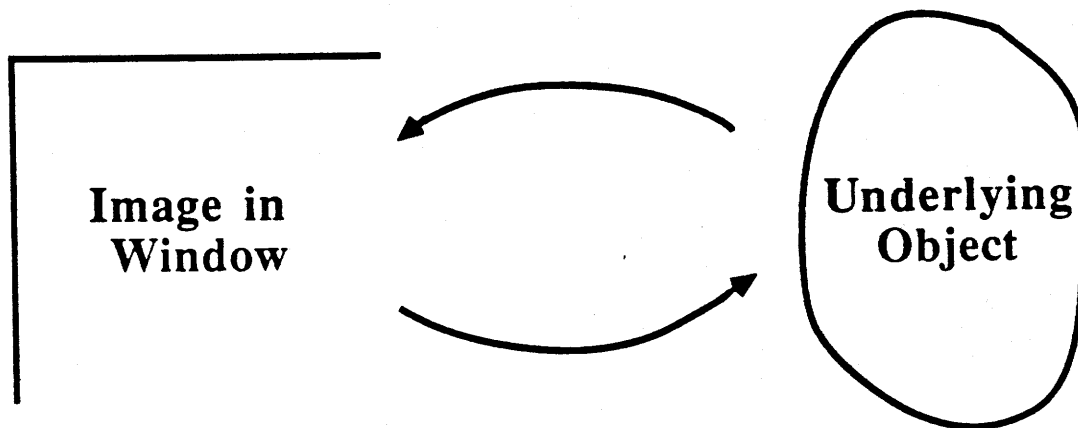
BANNER

= 'USERS PT_1 (library) world

begin
 [statement]

COMMAND WINDOW

Objects, Images, and Windows



- User sees an image, a representation, of the object in a window
- Image extends an arbitrary distance to the right and down
- Object must be updated to save changes in the image

Objects

- Environment is oriented around concept of an object
- Kinds of objects are
 - Text (file)
 - Ada
 - Library *world's + directories*
 - Other
- Specific form and structure are associated with each kind of object
- Environment knowledge of that form allows for object-specific editing operations
- Similar operations work across objects

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

- Environment Structure
- Environment Traversal
- Window Management
- Command Execution
- Help and Documentation
- General Editing

Ada Program Creation

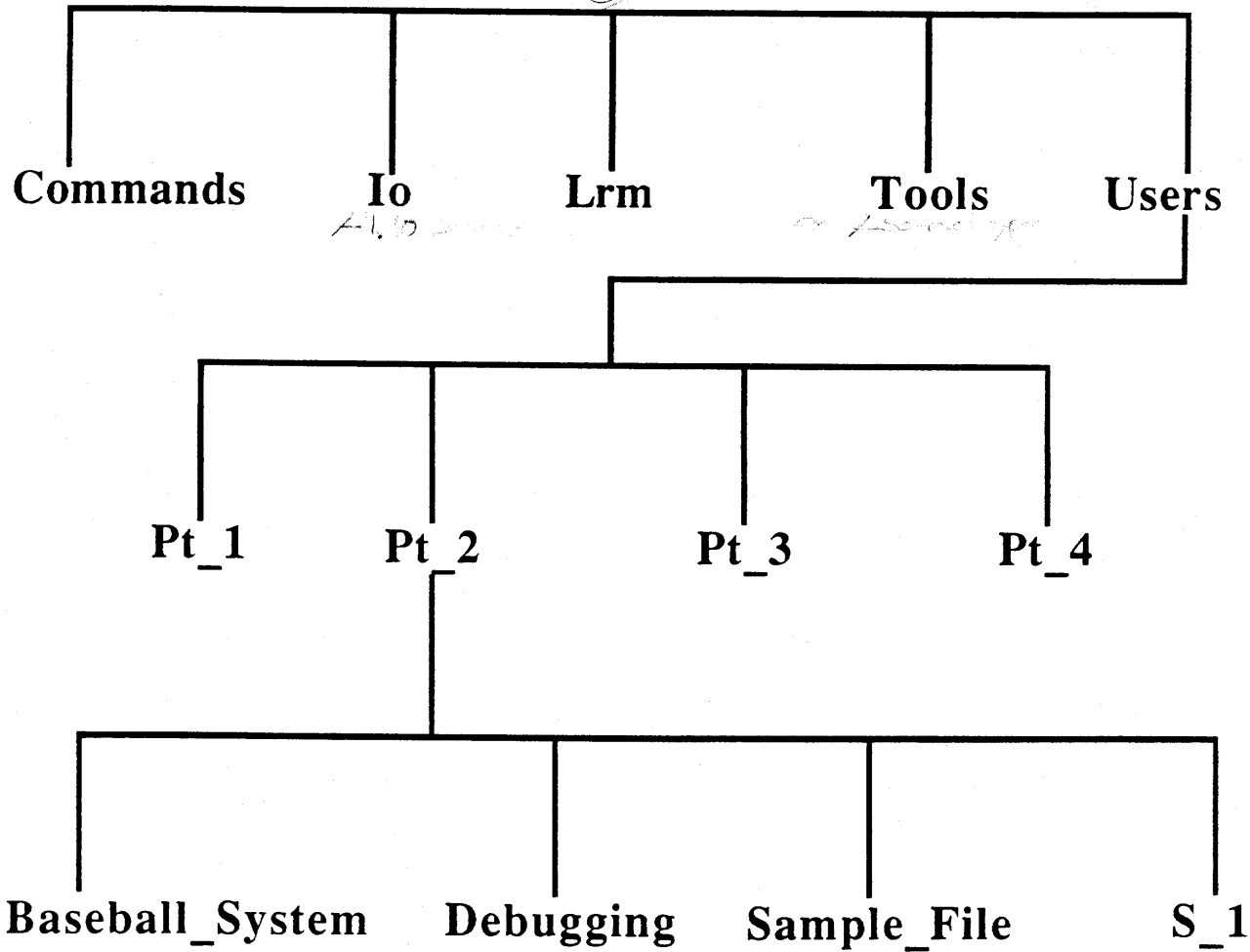
Ada Program Modification

Additional Topics

Components in Environment Structure

- Hierarchical structure of objects
- Classes of objects
 - Libraries (worlds and directories)
 - Ada units
 - Files
 - Others
- Root library is the world called “!”
- Arbitrary nesting of worlds and directories

Environment Hierarchy



Notation of Environment Structure

Root

```

!Library:
  Commands      : Library (World);
  Io             : Library (World);
  Lrm           : Library (World);
  Tools         : Library (World);
  Users         : Library (World);
= !library; world
    
```

subclass

```

!Users : Library:
  Pt_1         : Library (World);
  Pt_2         : Library (World);
  Pt_3         : Library (World);
  Pt_4         : Library (World);
= !USERS !library; world
    
```

CLASS *subclass*

```

!Users Pt_1 : Library (World):
  Baseball_System      : Library (World);
  Copyright_1986_Rational : File;
  Debugging           : Library (Directory);
  Experiment          : Library (World);
  Project_1          : Library (World);
  Rational_Commands  : Ada (Proc_Spec);
  Rational_Commands  : Ada (Proc_Body);
  Sample_File        : File;
  Statistics_System   : Library (World);
  S_1                : Session;
  S_1_Lost_Keys      : Pipe;
  S_1_Switches       : File (Switch);
= !USERS PT_1 !library; world
    
```

inner programming -- between SUGRA, COMPTON

```

!Users Pt_1 Baseball_System : Library (World):
  Baseball           : Ada (Pack_Spec);
  Baseball           : Ada (Pack_Body);
  Baseball_Statistics : Ada (Proc_Spec);
  Baseball_Statistics : Ada (Proc_Body);
  Data_Inputter      : Ada (Pack_Spec);
  Data_Inputter      : Ada (Pack_Body);
  Formatter          : Ada (Pack_Spec);
  Formatter          : Ada (Pack_Body);
= !USERS PT_1 BASEBALL_SYSTEM !library; world
    
```

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

Environment Structure

- Environment Traversal

Window Management

Command Execution

Help and Documentation

General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Moving to a New Object

- Called *traversing*
- Uses the hierarchical structure of the Environment
- Basic model
 - Point to the object of interest
 - Go to the object
- How to traverse
 - Move the cursor to the line containing the object
 - Go to the object: **Definition**

down in the hierarchy

the object of interest

Moving to a New Object, cont.

- Example

<u>!Users : Library;</u> Pt_1 : Library (World); Pt_2 : Library (World); Pt_3 : Library (World); Pt_4 : Library (World);	→	<u>!Users.Pt 4 : Library;</u> Baseball_System : Library (World); Debugging : Library (Directory); Sample_File : File; S_1 : Session;
--	---	--

- Additional commands

- Go to outer enclosing library structure or
Ada unit: `Enclosing Object`
- Go to user home world: `Home` `ESC` `↑`
- Move to upper window: `Window` - `↑`
- Move to lower window: `Window` - `↓`

Exercise: Traversing the Environment

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Start in your home world.
2. Move your cursor to the line containing the `Experiment` world.
3. Move your cursor to the line containing the `Project_1` world.
4. Go to the `Project_1` world.
5. Go to the `Code_Generator` world inside `Project_1`.
6. Go to the `Release_1` world inside `Code_Generator`.

Exercise: Traversing the Environment, cont.

7. Return to the window displaying the `Code_Generator` world.
8. Go to the world enclosing `Code_Generator`. You should be in `Project_1` after executing this step.
9. Go to the world enclosing `Project_1`. You should be back in your home world after executing this step.
10. Go to the world enclosing your home world. You should be in the `users` world after executing this step.
11. Go to the world enclosing `users`. You should be in the root world of the Environment, "!".
12. Return directly to your home world.

Exercise: Traversing the Environment, cont.

13. Go to the following places using the keys just explored. (Pt_n indicates your home world.)

- !Users.Pt_n.Project_1.Linker
- !Users.Pt_n.Project_1.Linker.Release_2
- !Users.Pt_n.Project_1.Code_Generator-
.Release_2
- !Users.Pt_n

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

Environment Structure

Environment Traversal

- Window Management

Command Execution

Help and Documentation

General Editing

Ada Program Creation

Ada Program Modification

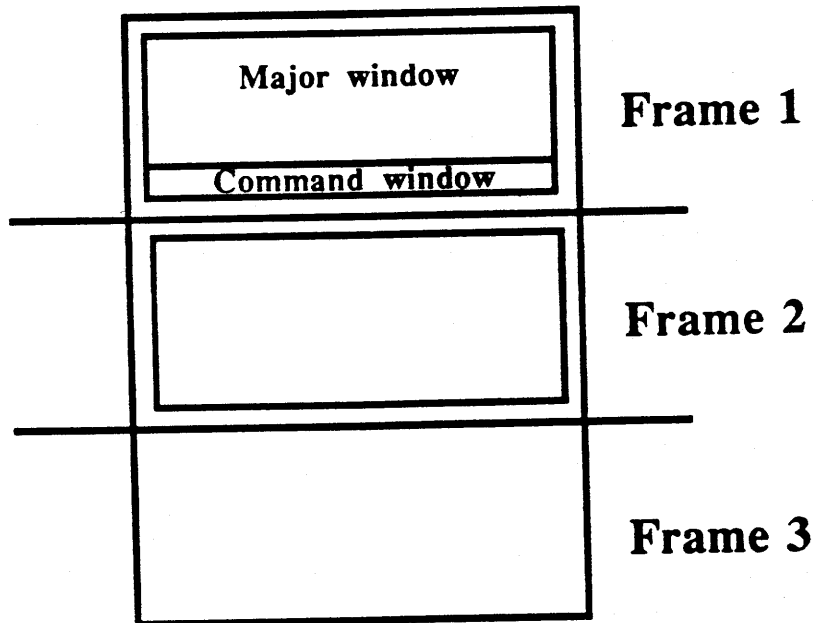
Additional Topics

Window Characteristics

- Environment creates windows
 - Overlays using least-recently-used algorithm
 - Marks next window to be overlaid with a tilde (~) in banner
ü
- User manages windows
 - Can alter the size and the location
 - Can explicitly remove
 - Can lock onto the screen

Screen Organization

- Screen is a set of *frames*
- Frame contains a major window plus any number of Command windows
- Environment default is to divide the screen into three frames



~ If the next window to be managed (over existing) is the banner

Window Manipulations

- **Managing the current window**
 - Cursor determines current window
 - Current window can be sized, moved, locked, or removed
- **Managing the Window Directory**
 - Window Directory contains set of all windows viewed
 - User can remove or view any window

Current Window Manipulations

- Sizing and positioning major windows

- Expand the current window size:

`Window` - `J`

- Transpose the current window and the window above:

`Window` - `T`

swap around!



- Locking major windows

- Lock the window in its current position:

`Window` - `Promote`

- Unlock the window: `Window` - `Demote`

`EDIT`

- Note that ^E@ in banner indicates the window is locked

Current Window Manipulations, cont.

- Removing windows

- Remove the window from the screen: *where my cursor is*

Window - **D**

- Remove the frame from the screen:

Window - **X**

- Removing images

- Release the object without saving the changes: *everything in the window*

Object - **G**

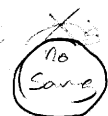
- Release the object and save the changes:

Object - **X**



*all changes
is not*

Window



*editor
release
no save*

Window Directory

- Keeps list of all images viewed and not released
- Window Directory replaces the next window to be overlaid
- Window Directory window is overlaid next
- Symbols
 - = means read-only image
 - (a blank) means read/write image and not changed since last save
 - * means image changed but not saved

Window Directory, cont.

• Example

= Logo PT_1 S_1 running

MOD	LINES	TYPE	LAST	BUFFER NAME
■	15	(text)	11:44:23 AM	..1.HISTORY_LOG_07_12_85'V(3)
■	4	(library)	11:44:15 AM	..1.CODE_GENERATOR.RELEASE_1
■	5	(library)	11:44:07 AM	..PT_1.PROJECT_1.CODE_GENERATOR
■	5	(library)	11:44:04 AM	!USERS.PT_1.PROJECT_1
■	14	(library)	11:39:47 AM	!USERS.PT_1
■	10		11:44:28 AM	Help Window
■	5		11:44:28 AM	Message Window
■	10	(windows)	11:44:28 AM	Window Directory

cursor
beginning

readOnly
read write access

more on the left

cursor
object G
deletes the window
⇒ price:
Selector
and...

= window Directory (windows)

* read write access changed but not saved

Window Directory Manipulations

- Display the Window Directory:

`Window` - `Definition`

- Display an image

- Align cursor on image of interest
(cursor keys, scrolling keys)

- View the image: `Definition`

Window Management Hints

- Use the Environment standard window sizing and placement
- Use `Window - J` to increase the size of major windows
- Use the window lock mechanism to retain on the screen windows of continual interest
- Remove windows from the screen when you no longer need them at the moment (you can always get them back by using the Window Directory)
- Remove images when you're through with them (this keeps your Window Directory smaller and makes it easier to find things quickly)

*terminal save process
show the...*

Exercise: Manipulating Windows

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

Start with three major windows on the screen. If you don't have three, use the Window Directory window to bring additional windows onto the screen. The screen is divided into three frames, numbered 1, 2, and 3 from the top of the screen.

1. Switch the arrangement of the windows so the major window in frame 1 is in frame 2.
2. Switch the arrangement of the windows so the major window in frame 3 is in frame 1. Notice where your cursor is positioned after the operation.

Window - T (change)

and the
go to process

Cursor moved

Exercise: Manipulating Windows, cont.

3. Expand the major window in frame 2 to occupy the space of frames 2 and 3. *window - J*
4. Using the Window Directory, go to the root of the Environment, "!" *then → [Environment]*
5. Lock the window displaying "!" on the screen. How can you tell if a window is locked on the screen? *[Window] [locked] E*
6. Using the Window Directory, go to `!Users.Pt_n.Project_1.Code_Generator-Release_1` *?*
7. Unlock the window displaying "!" *Window - [Environment]*
8. Notice which window will be overlaid. Go to `!Users.Pt_n.Project_1.Code_Generator-Release_1.History_Log_07_12_85.`

notice

[DEFINITION]

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

Environment Structure

Environment Traversal

Window Management

- Command Execution

Help and Documentation

General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Command Model

- Environment commands are Ada procedures
- Any Environment command can be executed from a Command window
- Common commands are bound to keys

Keys

- Procedures bound to keys execute identically from a Command window
- Users can bind any command to any key
- Additional information about any key is available through on-line help

On-Line Help on Keys

- **Command:** Help on Key
- **Command operation**
 - Prompts for key of interest in the Message window
 - Press key of interest
- **Help window**
 - Displays command name, key binding(s), brief summary from the Reference Manual
 - Can be scrolled

get into the window HELP window

from help window,

window - D

Command Window

- Displays Ada declare block where commands can be entered
- Can contain any arbitrary Ada code
- Example

```
!!Users_Pt_1_Baseball_System  
Baseball  
Baseball  
Baseball_Statistics  
Baseball_Statistics  
Data_Inputter  
Data_Inputter  
Formatter  
Formatter
```

```
= :USERS_PT_1_BASEBALL_SYSTEM |library| *world  
declare  
  use Editor, Library, Common;  
begin  
  [statement]  
end;
```

Command Window Operations

- Create a Command window from any window: **Create Command**
- Complete a command fragment including any parameters: **Complete** *see promote/sanitize*
- Execute a command: **Promote**
- Execute another command by entering the new command on the prompt
- Reexecute a command: **Promote**

Promote

- Means “I’m done - go do this operation”
- Is the most frequently used key
- Usage examples
 - Execute commands
 - Enter interactive I/O
 - Change state of Ada objects
 - Save text files

Prompts

- Characteristics

- Indicated by reverse video
- Disappear automatically as you type

- Usage

- Double quote remains for string parameters
- Turn off a prompt to modify the test:

~~Item Off~~ CTRL X

- Move between prompts: ~~Next Item~~,

Previous Item

ESC U

Exercise: Using Command Windows

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Create a Command window. *create command*
2. Enter `defin` and use the appropriate key to complete the command. What happened as you started typing on the prompt? *COMPLETE*
it disappeared
3. At the prompt, enter `!`. What happened to the prompt and the double quotes as you started typing? *show a view*
4. Execute the command. A window showing the root of the Environment appears! What happens to the command in the Command window? Remove the window displaying the root world, "`!`", from the screen. *REMOVE*
WINDOW
5. Return to the Command window and reexecute the same command. *WINDOW*
PROMPT

Exercise: Using Command Windows, cont.

6. Return to the Command window. Turn the command into text so you can modify the parameter.
7. Change the parameter to `!users` and reexecute the command.
8. Return to the Command window. Retype `defin` over the prompt and complete.
9. Move your cursor to the beginning of the line. Now move directly to the parameter (without using the cursor keys).

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

Environment Structure

Environment Traversal

Window Management

Command Execution

- Help and Documentation

General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Additional On-Line Help

- `Help on Help` key
- `Help on Command` key
 - Unambiguous names produce help message
 - Ambiguous names produce list of commands

Form of Help Messages

- Keybindings, if any, on first line
- Ada specification of command
- Description of command
 - May be longer than the size of Help window
 - Contains information from the Reference Manual

Documentation

- Reference Manual (5 volumes)
- Reference Summary
- Other manuals
 - System Manager's Guide
 - Networking Tools
 - Site Planning Guide
 - Terminal User's Manual
 - Project Management Manual

Indexes for the Documentation

- **Master Index**
 - Located in back of Reference Summary
 - Located in each Reference Manual volume
 - Contains alphabetical entries of packages, subprogram, types, and exceptions
- **Tables of Contents**
 - Located in front of each section in each volume
 - Contains normal Table of Contents
 - Contains alphabetical Table of Contents

Seminar Outline

Basic Mechanisms

Introduction

The Keyboard

The Screen

Environment Structure

Environment Traversal

Window Management

Command Execution

Help and Documentation

- General Editing

Ada Program Creation

Ada Program Modification

Additional Topics

Moving within an Image

- Commands provide movement by character, word, line, or image scrolling
- “Master Reference to Key Bindings by Topic” in the Keymap lists the commands under the “Moving within an Image” section

Editing Operations

- **Commands provide textual manipulation of characters, words, lines, or regions for common editing operations**
 - Deleting text
 - Moving and copying text
 - Transposing text
 - Controlling the case of text
 - Searching and replacing text
- **“Master Reference to Key Bindings by Topic” in the Keymap lists the commands under the “General Editing Operations” section**

Editing Operations: Search and Replace

- Four different operations

	Search	Replace
Forward	Control S ^F	ESC - F Meta S
Reverse	Control R	Meta R ESC - R

Handwritten notes:

Control F
↓
Control R
↓
Control F
↓
ESC - F FORWARD SEARCH
↓
ESC - R
↓
ESC - F
↓
ESC - R

Editing Operations: Search and Replace, cont.

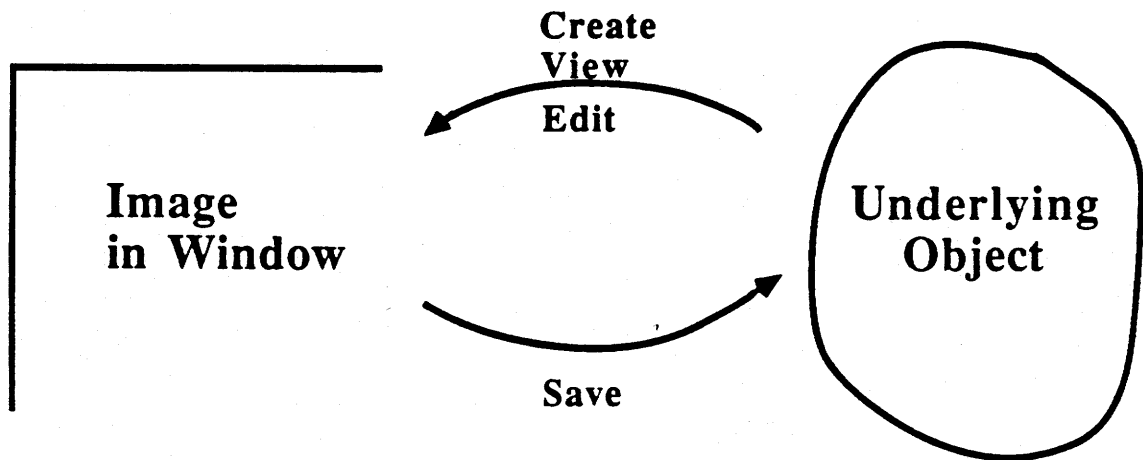
- Basic model
 - Search/replace command is initiated by pressing the search or replace key
 - Editor enters *composing* mode, which is indicated in the Message window banner
 - Search and replace strings are entered at Message window prompts
 - Composing is completed and command execution is started by pressing the search or replace key
 - Cursor is positioned one character after the target or replacement string (forward direction)
 - Replacement or finding of the next occurrence is done by pressing the search or replace key

Editing Operations: Search and Replace, cont.

- Additional commands

- Abort from composing mode: `Control G`
- Exit from search mode: `any key`
- Replace “n” occurrences: `Numeric n` prefix to the replace key
- Replace all occurrences: `Numeric -` - `Numeric 1` prefixes to the replace key

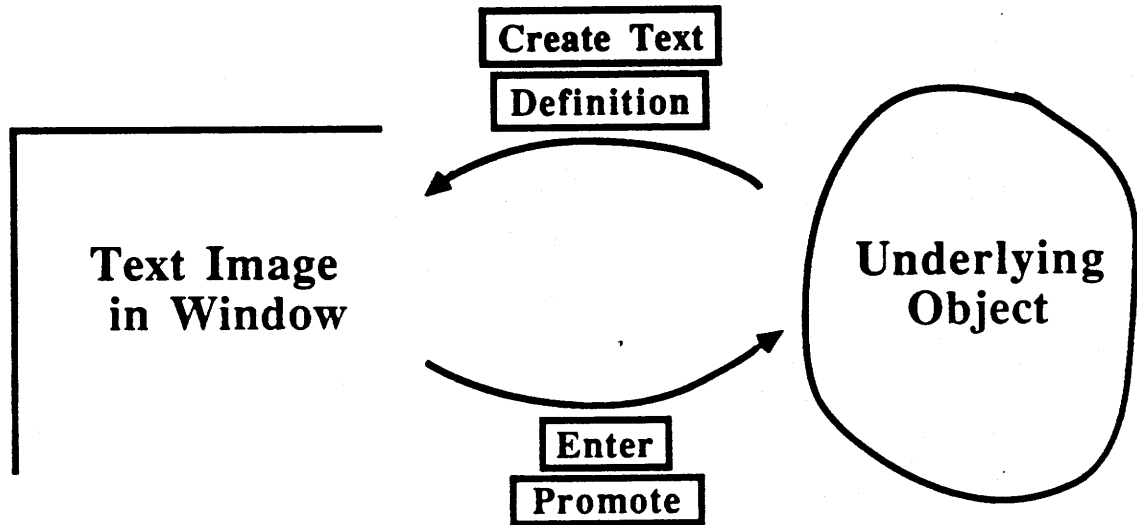
Basic Editing



- **Basic method**

- Retrieve the image of the underlying object through the create, view, or edit operations
- Enter the necessary editing changes
- Save the changes in the image to the underlying object

Text Objects



- **Commands**

- Create an object in a library: **Create Text**

- View the object with read-only access:

Definition

- View the object with read/write access:

Edit

- Save any changes to the object and retain read/write access: **Enter**

- Save any changes to the object and change to read-only access: **Promote**

Exercise: Editing Text

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Create a text file in your home world called `Alternate_Log`.

2. Locate and go to the image of the file `History_Log_07_12_85` in `!Users.Pt_1.Project_1.Code_Generator-Release_1`.

3. Copy the entire contents of this file into your newly created `Alternate_Log` file using region copy. Use the `Alternate_Log` file for the rest of the exercise.

4. Add the line `History log for the week of 07-12-85` to the beginning of the file.

1/20/83

Exercise: Editing Text, cont.

5. Use search to find the next occurrence of DBP.
6. Set the beginning of a region at this point.
7. Use search again to find `output`.
8. Set the end of the region at this point and copy it at the end of the file.
9. Save the file but retain read/write access so you can continue editing.
10. Go to the beginning of the file and kill the first line.
11. Move to the second line in this revised file.
12. Move two words to the right and delete the rest of the line.

Mike R

Exercise: Editing Text, cont.

13. Join the first two lines.
14. Search and replace all occurrences of the word `link` with the word `load`, except for the occurrence that is part of the word `Linked_List`.
15. Save the file and change the access from read/write to read-only.

Review

- What is on the screen?
- What are objects, images, and windows?
- What are common characteristics of windows?
- Describe the directory structure of the Environment.
- How do you move around the directory structure?
- How are commands invoked?
- What does `Promote` do?
- What is in the *Rational Environment Basic Operations* and when would you use it?

Review, cont.

- Where is the *Rational Environment Keymap*?
- What information does the on-line help system provide?
- What information would be found in the *Rational Environment Reference Summary*?

Seminar Outline

Basic Mechanisms

Ada Program Creation

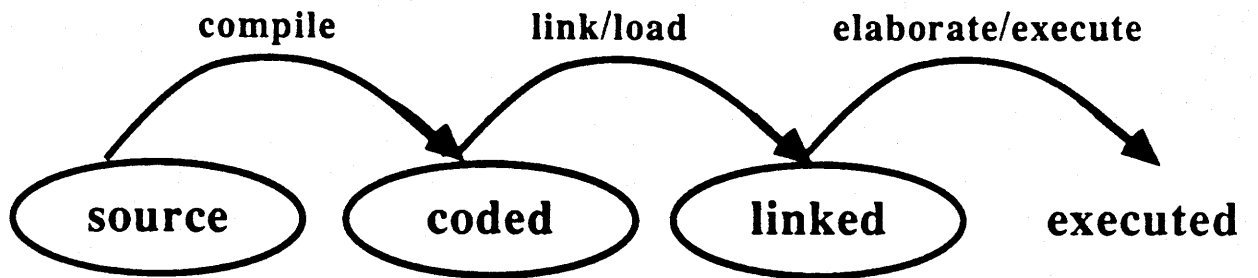
- Basic Concepts
- Ada Editing Aids
- Ada Units
- Unit Testing
- Organization of Ada Units
- More Ada Editing Aids
- Multiple-Unit Ada Programs

Ada Program Modification

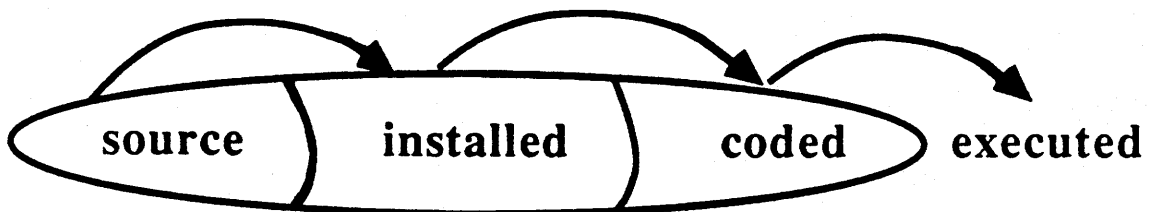
Additional Topics

Development Models

- Conventional model



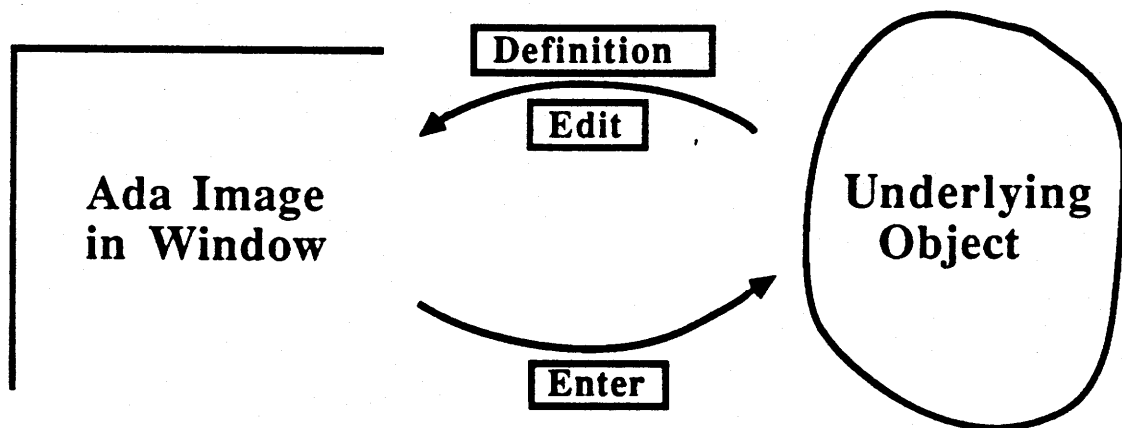
- Rational Environment model



Development Models, cont.

- Conventional model
 - Separate file for each “state”
 - Inconsistencies can exist between the actual source and what is being executed
 - Compilation management is manual and by convention only
- Rational Environment model
 - One object with “state” information
 - Actual image you see is what is being executed
 - Compilation is managed by the Environment

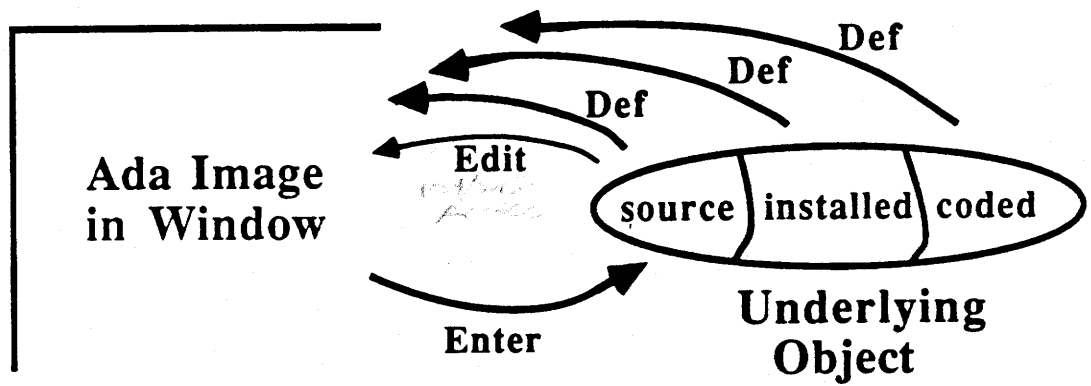
Ada Objects and Images



- View the Ada object with read-only access:
Definition
- View the Ada object with read/write access:
Edit
- Save the changes and retain read/write access: **Enter**

Ada Objects and Images, cont.

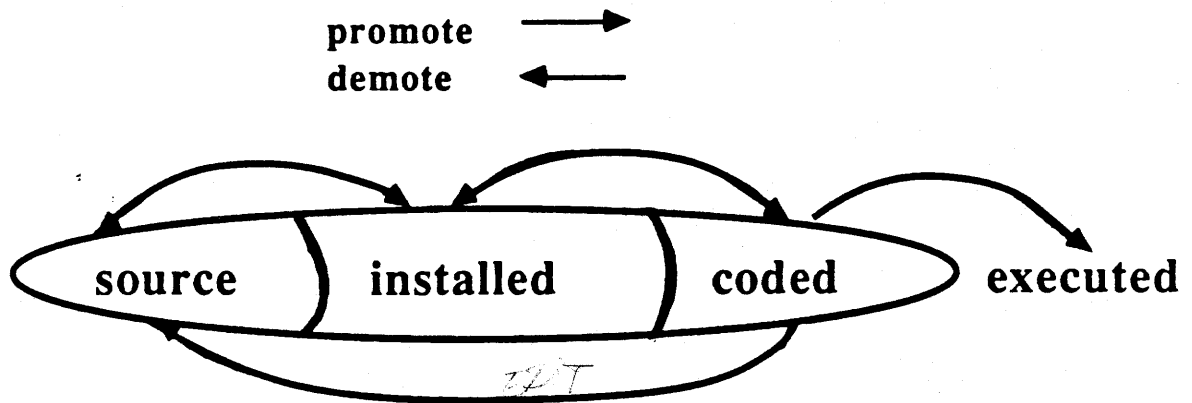
- Ada object expanded



Ada Object States

- Source
 - Editable
 - Not necessarily syntactically or semantically consistent
called
- Installed
 - Not editable but can be altered
 - Syntactically and semantically consistent
 - Can be referenced by other units
- Coded
 - Cannot be altered
 - Syntactically and semantically consistent
formal *semanticize*
 - Can be referenced by other units
 - Machine-code generated

Ada Object State Transitions



- Move the Ada object one state relative to the current state: `Promote`, `Demote`
- Move the Ada object to a specific state: `Source`, `Install`, `Code`, `Edit`
- Move entire libraries to a specific state with automated facilities

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

- Ada Editing Aids

Ada Units

Unit Testing

Organization of Ada Units

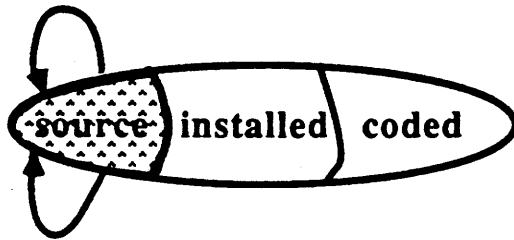
More Ada Editing Aids

Multiple-Unit Ada Programs

Ada Program Modification

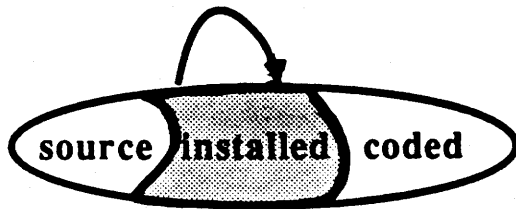
Additional Topics

Ada Object Editing



- **Format**
 - Incremental syntactic checking *synth. checking of what is in since last format.*
 - Syntactic completion
 - Pretty printing
- **Semanticize**
 - Incremental checking of Ada semantics

Ada Object Editing, cont.



Changes in installed state

- Incremental additions, changes, and deletions to statements and declarations allowed

Format

- Can be done on incomplete programs
- Checks for syntactic errors (for example, keyword in the wrong place)
- Provides minimum completion for incomplete code fragments
- Inserts prompts for required code additions
 - Move to the next prompt: ~~Next Item~~ ESC N
 - Move to the previous prompt: ~~Previous Item~~ ESC V
- Pretty prints the program

Format, cont.

- Example before Format

```
procedure format_example is t:integer
```

```
* [COMP_UNIT] |ada| source unit
```

- Example after Format

```
procedure Format_Example is
  T : Integer;
begin
  [statement]
end Format_Example;
```

```
# [COMP_UNIT] |ada| source unit
```

Semanticize

- Can be done on program fragments
- Verifies the meaning of Ada structures
- Example
 - Type incompatibility
 - Parameter profile matching
 - Declaration of named objects

Semanticize, cont.

- Example before Semanticize

```

= Logo Pt_10 S_1      running
-----
procedure Semanticize_Example is
  Defined : Boolean := True;
begin
  Undefined := Defined;
  Text_io.Put (Undefined);
end Semanticize_Example;

```

```

# (USERS Pt_10 _ADA_2_ v12) (ada) source unit

```

- Example after Semanticize

```

Semantic errors found
= Logo Pt_10 S_1      running
-----
procedure Semanticize_Example is
  Defined : Boolean := True;
begin
  Undefined := Defined;
  Text_io.Put (Undefined);
end Semanticize_Example;

```

```

# (USERS Pt_10 _ADA_2_ v12) (ada) source unit

```

Syntactic and Semantic Errors

- Message window displays error notification and explanations
- Each occurrence is underlined
- Commands

— Explain error further: Explain Item 03500-3

— Move to the next error: Next Item E-9-12

— Move to the previous error: Previous Item E-9-12 U

— Remove the current error designation:

Item Off C-104 X

Syntactic and Semantic Errors, cont.

- Example

```
Semantic errors found
UNDEFINED denotes no defined object or value ;
= Logo Pt_10 $_1      running
```

```
procedure Semanticize_Example is
  Defined : Boolean := True;
begin
  Undefined := Defined;
  Text_io.Put (Undefined);
end Semanticize_Example;
```

```
# !USERS Pt_10 _ADA_2_ v121 !ada! source unit
```

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

Ada Editing Aids

- Ada Units

Unit Testing

Organization of Ada Units

More Ada Editing Aids

Multiple-Unit Ada Programs

Ada Program Modification

Additional Topics

Ada Unit Creation

- Find (or create) a library
- Create a workspace in the library: **Object** - **I**
- Enter the program unit using incremental syntactic completion (**Format**) and semantic checking (**Semanticize**)
- Promote the Ada unit to the installed state when there are no semantic errors: **Promote**
- When the program unit is complete enough to run, promote to the coded state: **Promote**
- Execute the program by opening a Command window, entering the program name, and executing: **Promote**

Program Execution

- Program driver is executed from a Command window.
- Command procedure is elaborated and executed:
 - Run-time representation of your program is created (linked and loaded).
 - Ada elaboration of your program is completed.
 - Your program is executed.

p 85
p 27

Training Scripts

- Detailed instructions provided to accomplish specific tasks
 - Displayed in normal Environment windows
 - Can be overlaid by other windows
- Menu allows selecting one of several scripts
- Commands
 - Start up the scripts: **F1**
 - Return to the menu: **F2**
 - Return to previous step or previous menu item: **F3**
 - Go to next step or next menu item: **F4**
 - Select current menu item: **F5**

Training Scripts, cont.

- Example of script window

```
= Logo PT_1 S_1 ... running
```

Step 3. Find the procedure Print_Player_Stats in the package. This is...

~~Step 4. Locate and select the second statement in the procedure (the one with the_lines_At_Bat) by moving the cursor to that line and pressing [Object] - [*] repeatedly until the entire statement is highlighted (selected).~~

Step 5. Edit that statement by pressing [Edit]....

```
SCRIPT WINDOW (text) JOB 243 STARTED 12 01 05 PM Modifying Ada Program
```


Script: Creating Ada Programs

Use the “Creating Ada Programs” script to assist you.

Create and execute a program in a library that prints a “Hello World” message to the screen using format, semanticize, and promote.

Optional Exercise: Creating Ada Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Build a program to print out the phrase "My name is " and some name supplied as the program parameter. Create the program in the `Experiment` world in your home world. Use the specification provided below or one of your own.

```
procedure My_Name (The Name: String);
```

2. Enter the program using format and semanticize.
3. Execute the program and verify the result.

Exercise: Creating Ada Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Build a program to compute factorials in a library. Create it in the `Experiment` world in your home world.
2. Enter the program provided on the next page, or your own, using format and semanticize. It is possible to enter the program by pressing the `Space` key only 2 times. It is also possible to enter the program using the `Shift` key only 10 times. How close can you come?

Exercise: Creating Ada Units, cont.

```
with Text_IO;
procedure Factorial (N : Natural) is
    The_Result : Natural := 1;
begin
    for I in 1..N loop
        The_Result := The_Result * I;
    end loop;

    Text_IO.Put_Line
        (Natural'Image (N) & " ! " &
         Natural'Image (The_Result));
end Factorial;
```

3. Execute the program and verify the results.

Optional Exercise: Creating Ada Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Build a program to calculate the area of a circle, where the diameter is supplied as the program parameter, and print out the result. Create the program in the `Experiment` world in your home world.

The formula for the area of a circle is $\pi * \text{radius squared}$. The radius of a circle is $\text{diameter} / 2$. Use the specification provided below or one of your own.

```
procedure Area_Of_Circle (Diameter :  
Natural);
```

2. Enter the program using `format` and `semanticize`.
3. Execute the program.

Optional Exercise: Creating Ada Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Build a program to count the number of lines in a text file, where the filename is supplied as the program parameter, and print out the result. Create the program in the `Experiment` world in your home world.

Use the specification provided below or one of your own.

```
procedure Count_Lines (File_Name :  
String);
```

2. Enter the program using format and semanticize.
3. Input files have been provided in the `Experiment` world. Execute the program and verify the results. `Input1` has 270 lines. `Input2` has 416 lines.

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

Ada Editing Aids

Ada Units

- Unit Testing

Organization of Ada Units

More Ada Editing Aids

Multiple-Unit Ada Programs

Ada Program Modification

Additional Topics

Command Window Usage

- Executing Environment commands
- Executing user-created commands
- Executing Ada programs
- Testing and prototyping Ada units

Unit Testing with Command Windows

- Motivation: Provide mechanism to gain feedback on program algorithms
- Basic method
 - Create the Ada unit and promote it to the coded state *is done*
 - Create a Command window attached to the library containing the unit to be tested: `Create Command`
 - Enter the test program including any local variables and any calls to the unit:
use `Format` frequently
 - Check for semantic errors: use `Semanticize` frequently
 - Execute the test program: `Promote`

Script: Testing Ada Programs

Use the “Testing Ada Programs” script to assist you.

Test the `hello` program created in a previous script in a Command window.

Exercise: Testing Ada Programs

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Return to the `Experiment` library containing the `Factorial` program previously created.
2. Create a Command window and test your `Factorial` program for a range of values from 0 through 12. *take note*
3. Change your test program so that it will print a starting test message before the loop used in the previous step. Reexecute. *take note*

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

Ada Editing Aids

Ada Units

Unit Testing

- Organization of Ada Units
- More Ada Editing Aids
- Multiple-Unit Ada Programs

Ada Program Modification

Additional Topics

Motivation

- Ada systems consist of
 - Ada code
 - Documentation
 - Test drivers and data
- Libraries provide a means of organizing these components of a system

Libraries

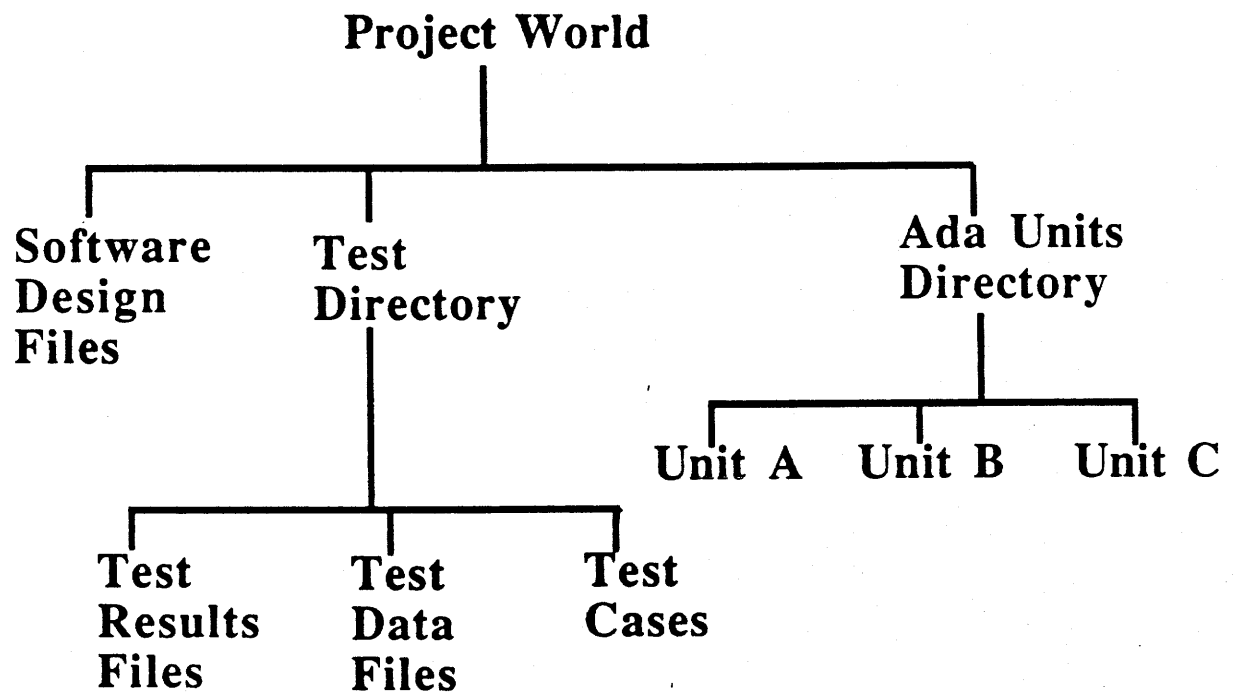
- Consist of two kinds
 - Worlds *to be a face to everything else*
 - Directories
- Are closed scope
 - Local units
 - Explicitly imported units

Kinds of Libraries

- **Worlds** typically structure systems at a project level, such as
 - Each user
 - Each project
 - Each major piece of a large project
- **Directories** organize the work within a project, such as
 - Documentation
 - Test data
 - Test scaffolds *simulates the external world for a new (test) director program*
 - Implementation of the project

Kinds of Libraries, cont.

- Example



Contents of Libraries

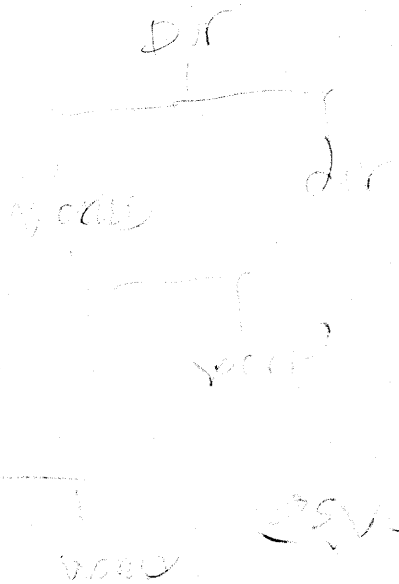
- Ada compilation units, such as procedures or packages
- Files (text objects) for documentation or test data
- Other libraries (worlds or directories) for further partitioning

Visibility in Libraries

- Units in “with” clauses must be either declared in or imported into the library
in the same world
- Units imported into the library use *links*
(via the same world)
- Utilizing a resource from another library is a two-step process
 - Import resources into the library via links
 - Import resources into the Ada unit via “with” clauses

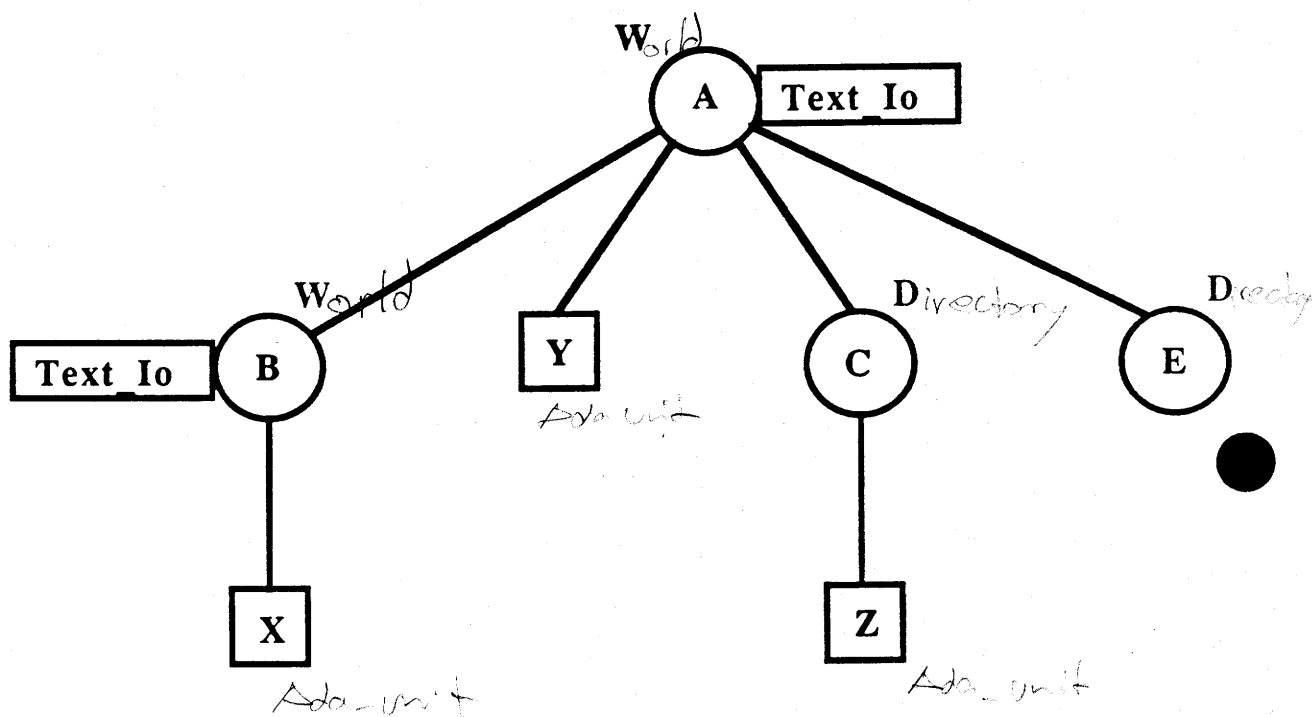
Links

- Import references to units outside a world
- Are associated with each world (not directories)
- Provide a mapping from simple Ada names to full library pathnames
- Are not inherited from enclosing worlds



Links, cont.

- Example



Evaluation of Context Clauses

- A unit that has been “withed” into another unit is searched for in two places in the following order
 - First look in the local library
 - Second look in the enclosing world’s set of links
- Absence of the necessary link is a common error

Internal Links

- Are a second kind of link
- Are created by default
- Provide visibility across directories within a world

Basic Link Operation

- View links: `Links.Display`

- Add one or more links:

`Links.Add("<full link name>");`

- Example

```
= Logo PT_1 $_1 running
```

```
!USERS.PT_1.EXPERIMENT % LINKS.DISPLAY STARTED 11:47:54 AM
```

86/06/26 11:47:56 --- Links from !USERS.PT_1.EXPERIMENT.

Link	Kind	Actual Name
FACTORIAL	INT	!USERS.PT_1.EXPERIMENT.FACTORIAL
HELLO	INT	!USERS.PT_1.EXPERIMENT.HELLO
TEXT_10	EXT	!10.TEXT_TO

! root world

```
!USERS.PT_1.EXPERIMENT % LINKS.DISPLAY !TEXT!
```

```
!Users.Pt_1.Experiment : Library (World):
```

```
Factorial : Ada (Proc_Spec);
Factorial : Ada (Proc_Body);
Hello : Ada (Proc_Spec);
Hello : Ada (Proc_Body);
Input1 : File;
Input2 : File;
```

*EXT:
EXTERNAL
FOR MY
WORLD*

```
= !USERS.PT_1.EXPERIMENT !library! ~ world
```

```
begin
```

```
Links Display
```

*Rational for better with the unit
set of the ad program and some
name seen in link!*

Exercise: Links

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you. You will need to look up the full pathname of `Text_10` in other documentation.

1. Go to the world called `Statistics_System` in your home world.
2. List the links for the `Statistics_System` world. (There should be none.)
3. Add a link for `Text_10`. Where can you find the pathname for `Text_10`?
4. List the links for the `Statistics_System` world again.

Exercise: More Links

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you. You will need to look up the full pathname for `Set_Generic` and `List_Generic` in other documentation.

1. Go to the world called `Experiment` in your home world.
2. List the links for the `Experiment` world. (There should be internal and external links.)
3. Add a link for `Set_Generic`.
4. Add a link for `List_Generic`.
5. List the links for the `Experiment` world again. Notice the new external links.

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

Ada Editing Aids

Ada Units

Unit Testing

Organization of Ada Units

- More Ada Editing Aids
- Multiple-Unit Ada Programs

Ada Program Modification

Additional Topics

Edit Histories

- Chain of editing changes is retained
- Each format operation creates a new entry on the chain
- Step backward through chain: `Object` - `U`
- Step forward through chain: `Object` - `R`

Chain object of history

Template Generation

- Uses semantic information about the object and knowledge of Ada
- Constructs and prompts for completion of program unit specification: `Create Body Part`
- Constructs and prompts for completion of a package private part: `Create Private Part`

Managing the Compilation of the System

- When the system is complete enough to run or unit test, use the automated facility to promote all units to the coded state:

`Compilation Make`

- Manages all compilation dependencies
- Ensures that the semantic consistency of the system is maintained

Using the Automated Compilation Facility: Make

- Promotes all units of an Ada system to the coded state following Ada compilation rules:

`Compilation Make`

- Sends output logs to the standard output window
- Log symbols

— `===` means current status info

— `+++` means forward progress

— `---` means commentary

— `...` means message continuation

— `++*` means an error occurred

— `***` means an error occurred

Using the Automated Compilation Facility: Make, cont.

• Example

```

=====
= Logo PT_1 $..1 .. running
=====
-----
!USERS.PT_1.STATISTICS_SYSTEM % COMPILATION.MAKE          STARTED 11:53:50 AM
-----
86/06/26 11:53:51 === [Compilation.Promote ("", ALL_PARTS, CODED, SAME_WORLD,
86/06/26 11:53:52 ... FALSE, PERSEVERE);].
86/06/26 11:53:52 +++ !USERS.PT_1.STATISTICS_SYSTEM.CALCULATE_STATS has been
86/06/26 11:53:52 ... INSTALLED.
86/06/26 11:53:54 +++ !USERS.PT_1.STATISTICS_SYSTEM.FLOAT_STATISTICS has been
86/06/26 11:53:54 ... INSTALLED.
86/06/26 11:53:55 +++ !USERS.PT_1.STATISTICS_SYSTEM.INTERFACE has been
86/06/26 11:53:55 ... INSTALLED.
86/06/26 11:53:58 +++ !USERS.PT_1.STATISTICS_SYSTEM.CALCULATE_STATS'BODY has
86/06/26 11:53:58 ... been INSTALLED.
86/06/26 11:54:07 +++ !USERS.PT_1.STATISTICS_SYSTEM.FLOAT_STATISTICS'BODY has
86/06/26 11:54:07 ... been INSTALLED.
86/06/26 11:54:13 +++ !USERS.PT_1.STATISTICS_SYSTEM.INTERFACE'BODY has been
86/06/26 11:54:13 ... INSTALLED.
86/06/26 11:54:14 +++ !USERS.PT_1.STATISTICS_SYSTEM.CALCULATE_STATS has been
86/06/26 11:54:14 ... CODED.
86/06/26 11:54:15 +++ !USERS.PT_1.STATISTICS_SYSTEM.FLOAT_STATISTICS has been
86/06/26 11:54:15 ... CODED.
86/06/26 11:54:15 +++ !USERS.PT_1.STATISTICS_SYSTEM.INTERFACE has been CODED.
86/06/26 11:54:17 --- Messages generated while promoting !USERS.PT_1.
86/06/26 11:54:17 ... STATISTICS_SYSTEM.CALCULATE_STATS'BODY to CODED.
86/06/26 11:54:17 ---      60 instructions for subprog CALCULATE_STATS.
86/06/26 11:54:17 ---      149 instructions for segment 990466.
86/06/26 11:54:17 +++ !USERS.PT_1.STATISTICS_SYSTEM.CALCULATE_STATS'BODY has
86/06/26 11:54:17 ... been CODED.
86/06/26 11:54:20 --- Messages generated while promoting !USERS.PT_1.
86/06/26 11:54:20 ... STATISTICS_SYSTEM.FLOAT_STATISTICS'BODY to CODED.
86/06/26 11:54:20 ---      167 instructions for package FLOAT_STATISTICS.
86/06/26 11:54:20 ---      300 instructions for segment 991490.
86/06/26 11:54:21 +++ !USERS.PT_1.STATISTICS_SYSTEM.FLOAT_STATISTICS'BODY has
86/06/26 11:54:21 ... been CODED.
86/06/26 11:54:24 --- Messages generated while promoting !USERS.PT_1.
86/06/26 11:54:24 ... STATISTICS_SYSTEM.INTERFACE'BODY to CODED.
86/06/26 11:54:24 ---      133 instructions for package INTERFACE.
86/06/26 11:54:24 ---      317 instructions for segment 992514.
86/06/26 11:54:24 +++ !USERS.PT_1.STATISTICS_SYSTEM.INTERFACE'BODY has been
86/06/26 11:54:24 ... CODED.
86/06/26 11:54:24 +++ 6 units were INSTALLED.
86/06/26 11:54:24 +++ 6 units were CODED.
86/06/26 11:54:24 === [End of Compilation.Promote Command].
=====
STATISTICS_SYSTEM % COMPILATION.MAKE (text)
=====

```

Seminar Outline

Basic Mechanisms

Ada Program Creation

Basic Concepts

Ada Editing Aids

Ada Units

Unit Testing

Organization of Ada Units

More Ada Editing Aids

- Multiple-Unit Ada Programs

Ada Program Modification

Additional Topics

Creating Ada Systems—Basic Method

- Create the set of specifications for the components of the system
- Complete the implementation for all components of the system
- Verify system functions

Building the Specifications

- Find or create a library and set up the necessary links
- Create a workspace in the library to enter the spec for an Ada unit: `Object` - `I`
- Enter each specification using incremental syntax completion (`Format`) and semantic checking (`Semanticize`)
- As each specification is completed, install the specification: `Promote`

Completing the Implementation

- Use the automated facility to create a skeletal body for each unit: `Create Body Part`
- Enter the implementation for each body using incremental syntax completion (`Format`) and semantic checking (`Semanticize`)
- As each body is completed, install the body: `Promote`
- Test completed units as appropriate using automated compilation facilities and Command windows

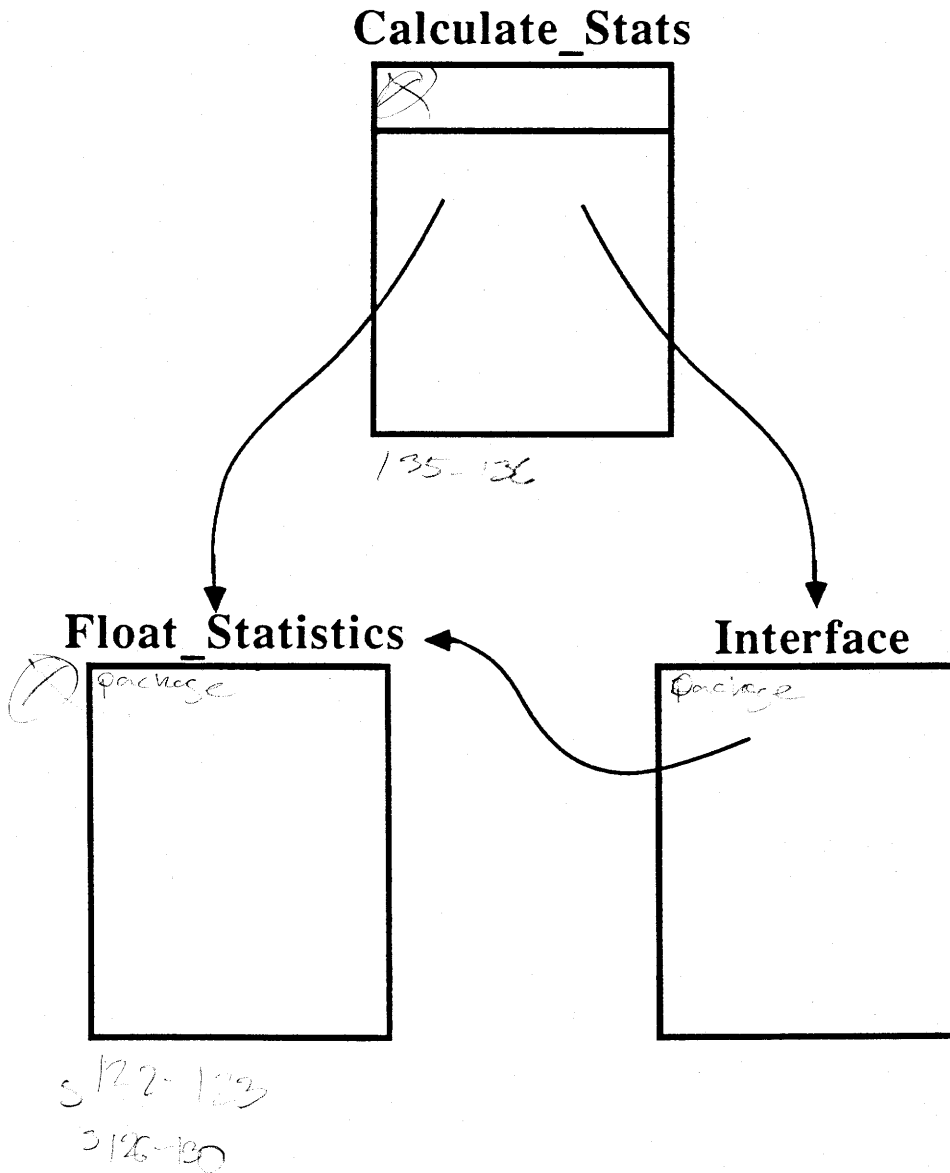
Exercise: Creating Ada Systems

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you in the next series of exercises on creating Ada systems.

Build a `calculate_stats` program that queries the user for raw input values and displays the mean, median, and range of those values.

Exercise: Creating Ada Systems, cont.

- Calculate_Stats program



Exercise: Creating Ada Systems— Build Specification of Float_Statistics

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Go to the `statistics_System` world in your home world. This will be the library used for the series of exercises using the `Calculate_Stats` program.
2. Enter the specification for the `Float_Statistics` into the library using `format` and `semanticize`. The code is provided on the next page.

Strive to minimize your keystrokes through the use of the `format` operation.

Exercise: Creating Ada Systems— Build Specification of Float_Statistics, cont.

```
package Float_Statistics is
    type Values is array (Integer range <>) of Float;
    function Smallest (The_Values : Values) return Float;
    function Largest (The_Values : Values) return Float;
    function Mean (The_Values : Values) return Float;
    function Median (The_Values : Values) return Float;
end Float_Statistics;
```

3. Promote the specification when it is complete.

Exercise: Creating Ada Systems— Build Specification of Interface

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Go to the `Statistics_System` world in your home world.
2. Enter the specification for the `Interface` into the library using `format` and `semanticize`. The code is provided on the next page.

Strive to minimize your keystrokes through the use of the `format` operation.

Exercise: Creating Ada Systems— Build Specification of Interface, cont.

```
with Float_Statistics;  
package Interface is  
  
    procedure Get (The_Values : out Float_Statistics.Values);  
    procedure Put (The_Header : String;  
                  The_Result : Float);  
    procedure Put (The_Header : String;  
                  The_Values : Float_Statistics.Values);  
  
end Interface;
```

3. Promote the specification when it is complete.

Exercise: Creating Ada Systems— Complete Implementation of Float_Statistics

1. Enter the package body for `Float_Statistics` into the library. Use the code provided on the following pages.

Use the create body part, format, and region copy operations judiciously to minimize the number of keystrokes.

Exercise: Creating Ada Systems— Complete Implementation of Float_Statistics, cont.

```
package body Float_Statistics is
  procedure Sort (The_Values : in out Values) is
    Switch_Value : Float;
    Is_Sorted    : Boolean := False;
  begin
    while not Is_Sorted loop
      Is_Sorted := True;

      for Index in The_Values'First..The_Values'Last - 1 loop
        if The_Values (Index) > The_Values (Index + 1) then
          Switch_Value := The_Values (Index + 1);
          The_Values (Index + 1) := The_Values (Index);
          The_Values (Index) := Switch_Value;
          Is_Sorted := False;
        end if;
      end loop;
    end loop;
  end Sort;
end Float_Statistics;
```

Exercise: Creating Ada Systems— Complete Implementation of Float_Statistics, cont.

```
function Smallest (The_Values : Values) return Float is
  Smallest_Value : Float := The_Values (The_Values'First);
begin
  for Index in The_Values'Range loop
    if The_Values (Index) < Smallest_Value then
      Smallest_Value := The_Values (Index);
    end if;
  end loop;

  return Smallest_Value;
end Smallest;
```

```
function Largest (The_Values : Values) return Float is
  Largest_Value : Float := The_Values (The_Values'First);
begin
  for Index in The_Values'Range loop
    if The_Values (Index) > Largest_Value then
      Largest_Value := The_Values (Index);
    end if;
  end loop;

  return Largest_Value;
end Largest;
```

Exercise: Creating Ada Systems— Complete Implementation of Float_Statistics, cont.

```
function Mean (The_Values : Values) return Float is
  Sum : Float := 0.0;
begin
  for Index in The_Values'Range loop
    Sum := Sum + The_Values (Index);
  end loop;
  return Sum / Float (The_Values'Length);
end Mean;

function Median (The_Values : Values) return Float is
  Sorted_Values :
    Values (The_Values'First..The_Values'Last) :=
    The_Values;
begin
  Sort (Sorted_Values);
  return Sorted_Values ((The_Values'Length / 2) + 1);
end Median;

end Float_Statistics;
```

Exercise: Creating Ada Systems— Complete Implementation of Float_Statistics, cont.

2. Promote the package body when complete.

3. Unit test the body of `Float_Statistics`.

Use the automated facility to promote all the units in your system to the coded state.

4. Create a test program in a Command window. A sample test is provided below. In turn, test the `Smallest`, `Largest`, and `Median` functions.

You should be able to test each function by simply modifying the prompt.

```
declare
    Max_Size      : Natural := 5;
    The_Values    : Float_Statistics.Values (1..Max_Size) :=
        (7.0, 5.0, 1.0, 2.0, 1.0);
    package Flt_Io is new Text_Io.Float_Io (Float);
begin
    Flt_Io.Put (Float_Statistics.Smallest (The_Values));
end;
```

WIKI TEST HERE

Exercise: Creating Ada Systems— Complete Implementation of Interface

1. Enter the package body for `Interface` into the library. Use the code provided on the following pages.

Use the create body part, format, and region copy operations judiciously to minimize the number of keystrokes.

Exercise: Creating Ada Systems— Complete Implementation of Interface, cont.

```
with Float_Statistics;
with Text_Io;
package body Interface is

    package Tio renames Text_Io;
    package Fio is new Tio.Float_Io (Float);

    procedure Get (The_Values : out Float_Statistics.Values) is
    begin
        Tio.Put_Line
            ("Enter" & Integer'Image (The_Values'Last) &
             " floating point values one at a time.");
        Tio.Put_Line ("Press ENTER after each value.");

        for Index in The_Values'Range loop
            loop
                begin
                    Tio.Put ("Type input #"
                            & Integer'Image (Index) & ": ");
                    Fio.Get (The_Values (Index));
                    exit;
                exception
                    when Tio.Data_Error =>
                        Tio.Put_Line
                            ("Invalid input value. Try again.");
                        Tio.Skip_Line;
                end;
            end loop;
        end loop;
    end Get;
```


Exercise: Creating Ada Systems— Complete Implementation of Interface, cont.

```
procedure Put (The_Header : String;
              The_Result : Float) is
begin
  Tio.New_Line;
  Tio.Put (The_Header & " ");
  Fio.Put (The_Result,
          Fore => 0, Aft => 2, Exp => 0);
  Tio.New_Line;
end Put;

procedure Put (The_Header : String;
              The_Values : Float_Statistics.Values) is
begin
  Tio.New_Line (2);
  Tio.Put (The_Header & " ");
  for Index in The_Values'Range loop
    Tio.New_Line;
    Fio.Put (The_Values (Index),
            Fore => 0, Aft => 2, Exp => 0);
  end loop;
  Tio.New_Line (2);
end Put;
end Interface;
```

Exercise: Creating Ada Systems— Complete Implementation of Interface, cont.

2. Promote the package body when complete.
3. Test your `Interface` package to see if it is receiving the correct input values. Use the test program provided or one of your own.
4. Use the automated facility to promote all the units in your system to the coded state.
5. Create a test program in a Command window. A sample test is provided below.

```
declare
    Max_Size    : Natural := 10;
    The_Values  : Float_Statistics.Values (1..Max_Size);
begin
    Interface.Get (The_Values);
    Interface.Put (The_Values);
    ("SAMPLE TEST CASE", THE_VALUES);
end;
```

6. Execute your test.

Exercise: Creating Ada Systems— Create Main Program

1. Enter the main program unit, `calculate_stats`, into the library. Use the code provided on the following pages.

Use the format and region copy operations judiciously to minimize the number of keystrokes.

Exercise: Creating Ada Systems— Create Main Program, cont.

```
with Float_Statistics,
    Interface;
procedure Calculate_Stats (Number_Of_Values : Natural) is
    The_Values :
        Float_Statistics.Values (1..Number_Of_Values) :=
            (others => 0.0);
begin
    Interface.Get (The_Values);
    Interface.Put ("The input data values are", The_Values);

    Interface.Put ("The range is",
        Float_Statistics.Largest (The_Values) -
        Float_Statistics.Smallest (The_Values));
    Interface.Put ("The mean is",
        Float_Statistics.Mean (The_Values));
    Interface.Put ("The median is",
        Float_Statistics.Median (The_Values));
end Calculate_Stats;
```

2. Promote the main program unit when complete.

Exercise: Creating Ada Systems— Verify the System

1. Use the automated facility to ensure that all units are in the coded state.
2. Execute the system.
3. The program prompts you for data. Enter some values as requested to demonstrate that the program works.

Each data value entered is terminated with

`Promote`.

Review

- What do libraries contain and for what are they used?
- What is the visibility in libraries?
- How do you import resources into libraries?
- What is object state?
- What do formatting and semanticizing do?
- In what ways can you transition Ada units from one state to another?
- How many objects exist for an Ada unit?
- How are Ada units added to libraries?

Review, cont.

- How do you execute Ada programs?
- How can you unit-test program units?
- How would you find out how to create and execute an Ada program?
- How and where would you find out about other commands for Ada objects?

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

- Simple Browsing
- Introduction to the Debugger
- Program Modification—Single-Unit Method
- Program Modification—Multiple-Unit Method

Additional Topics

Simple Browsing

- Motivation

- What is the exact type definition for a program variable
- What is the definition of a subprogram
- Where is a subprogram defined

- Commands

- Move to the enclosing Ada unit or library:

Enclosing Object

OP in the window

- Move from unit specs to bodies and vice

versa: **Ada Other Part**

Body → SPEC SPEC → Body

- View the definition of a *selected* structure:

Definition

OBJECT → Ada Other Part




Selection for Browsing

- Based on structure of Ada programs
- Examples
 - Select entire package or subprogram in a package body
 - Select single declaration or type components of declarations
 - Select entire statement or specific portions of nested statements
 - Select subprogram calls or any parameters
 - Select unit in “with” clauses

Selection

- Specifies the object on which the command will operate
- Examples
 - What object to view
 - What object to modify
 - What object to delete
 - What object to move
 - What object to display the value of in the Debugger

Selection Commands

- Select the structure at the cursor: **Object** - 
- Select successively larger structures:
Object -  *to select the structure*
- Select successively smaller structures:
Object - 

Selection Commands, cont.

- Example

- Initially

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
  end;
end loop;

```

- After using -

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
  end;
end loop;

```

} Blok

Selection Commands, cont.

— After using -

```
loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
  end;
end loop;
```

— After using -

*change of class definition
for member's project*

```
loop
declare
Current_Player : Baseball.Player_Statistics;
  begin
Data_Inputter.Get_Record (Current_Player);
Baseball.Percentage (Current_Player);
  end;
end loop;
```

Selection Commands, cont.

- Example

- Initially

```
Current Player: Baseball Player Statistics
```

- After using `Object` - `→`

```
Current Player: Baseball Player Statistics;
```

- After using `Object` - `←`

```
Current Player: Baseball Player Statistics
```

Selection Commands, cont.

- Select the next structure: -
- Select the previous structure: -

Selection Commands, cont.

- Example

- Initially

```
begin
Data_Inputter_Get_Record (Current_Player);
  Baseball.Percentage (Current_Player);
  Baseball.Sum (Current_Player, Team_Sums);
  Baseball.Add (Current_Player, Team_Statistics);
end;
```

- After using -

```
begin
  Data_Inputter_Get_Record (Current_Player);
Baseball.Percentage (Current_Player);
  Baseball.Sum (Current_Player, Team_Sums);
  Baseball.Add (Current_Player, Team_Statistics);
end;
```

- After using -

```
begin
Data_Inputter_Get_Record (Current_Player);
  Baseball.Percentage (Current_Player);
  Baseball.Sum (Current_Player, Team_Sums);
  Baseball.Add (Current_Player, Team_Statistics);
end;
```

Selection Commands, cont.

- Example

— Initially

```
Baseball : Baseball.Player_Statistics;
```

— After using `Object` - `↓`

```
Current_Player : Baseball.Player_Statistics;
```

- Example

— Initially

```
Baseball.Add (Current_Player, Team_Statistics);
```

— After using `Object` - `↓`

```
Baseball.Add (Current_Player, Team_Statistics);
```

— After using `Object` - `↓` again

```
Baseball.Add (Current_Player, Team_Statistics);
```

Selection Commands, cont.

- Select several structures above or below:

- - OR - -

- Example

— Initially

```
begin
Data_Inputter.Get_Record (Current_Player);
  Baseball.Percentage (Current_Player);
  Baseball.Sum (Current_Player, Team_Sums);
  Baseball.Add (Current_Player, Team_Statistics);
end;
```

— After using - -

```
begin
  Data_Inputter.Get_Record (Current_Player);
  Baseball.Percentage (Current_Player);
  Baseball.Sum (Current_Player, Team_Sums);
Baseball.Add (Current_Player, Team_Statistics);
end;
```

Exercise: Using Selection in Ada Units

Use the *Rational Environment Keymap* and your notes to assist you.

1. Go to the `Baseball_System` world in `!Users.Pt_n`.
2. View the body of the `Baseball_Statistics` unit. Remember that the body is the second instance of the unit name in a library. This program calculates individual team batting statistics. It prompts the user for input about players, such as the number of times at bat, number of hits, and number of runs batted in. It then calculates and displays batting percentages and team totals.

Exercise: Using Selection in Ada Units, cont.

3. Using the selection operations and without explicitly moving the cursor, in turn select:
 - The entire context clause.
 - The entire procedure `Baseball_Statistics` with its context clause.
 - Just the context clause again.
 - The entire procedure `Baseball_Statistics` without its context clause.
 - Just the name of the procedure (`Baseball_Statistics`).
 - The entire first object declaration (`Team_Sums : ...`).
 - Just the name of the first object declared (`Team_Sums`). *Object →*

Exercise: Using Selection in Ada Units, cont.

- Just the type of the first object declared
(`Baseball.Total_Players_Statistics`).
- The whole first object declaration again.
- The first statement (`Baseball.Init_Team_Stats`).
- The entire `Baseball_Statistics`
procedure.
- Just the first statement again
(`Baseball.Init_Team_Stats`).
- The begin block below the first state-
ment (`begin ... exception ... end;`).
- The statement below the begin block
(`Baseball.Percentage (Team_Sums)`).
- The begin block below the first state-
ment again (`begin ... exception ... end;`).

Exercise: Using Selection in Ada Units, cont.

- The entire loop (`loop ... end loop;`).
- The declare block in the loop (`declare ... begin ... end;`).
- The entire object declaration in the declare block (`Current_Player : ...`).
- The first statement in the declare block (`Data_Inputter.Get....`).
- The second statement in the declare block (`Baseball.Percentage (...)`).
- The subprogram name in the second statement without its parameters (`Baseball.Percentage`).
- The subprogram parameter without the subprogram name (`Current_Player`).
- The next entire statement in the declare block (`Baseball.Sum (.....)`).

Exercise: Using Selection in Ada Units, cont.

- The first parameter of the subprogram call (`Current_Player`).
- The second parameter of the subprogram call (`Team_Sums`).
- The first parameter again.
- The subprogram name without its parameters (`Baseball.Sum`).
- The entire while loop near the end of the procedure (`while ... loop ... end loop;`).
- The conditional expression in the while loop (`not Baseball.Is_Done (...)`).

Exercise: Using Selection in Ada Units, cont.

- The subprogram call without its parameters (`Baseball.Is_Done`).
- Just the subprogram parameter (`Player_Iterator`).
- The entire conditional expression again.
- The last statement in the procedure (`Formatter.Print...`).

Optional Exercise: More Selection in Ada Units

Use the *Rational Environment Keymap* and your notes to assist you.

1. Go to the `Baseball_System` world in `!Users.Pt_n`.
2. View the specification of the `Baseball` unit.
3. Using the selection operations and without explicitly moving the cursor, in turn select:
 - Just the context clause.
 - The entire package without the context clause.
 - Just the name of the package.

Optional Exercise: More Selection in Ada Units, cont.

- The third subtype (`Number_Hits`). Use a numeric prefix to get there with only three keystrokes.
- The entire record type for `Player_Statistics`. Again use a numeric prefix.
- Just the name of the type (`Player_Statistics`).
- The first record component of the type (`The_Name : ... := ...;`).
- Just the name of the first component (`The_Name`).
- The type of the first component (`Name`).

Optional Exercise: More Selection in Ada Units, cont.

- The initialization of the first component (`(others => ' ');`).
- The entire third component of the type (`The_Number_Hits : ... := 0;`). Use a numeric prefix.
- The entire record type for `Total_Players_Statistics`. Use a numeric prefix.
- Just the type of the third record component (`Total_Runs_Batted_In : ... := 0;`).
- The initialization of the third record component. This will fail. You can't select a literal value.
- The package `Team` in the private part.

Exercise: Browsing Ada Systems

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Go to the `Baseball_System` world in `!Users.Pt_n`.
2. Notice the units that make up this system.
3. Get the definition of the procedure body of the `Baseball_Statistics` program.
4. Look at the declarative region of `Baseball_Statistics`. Three types are used: `Total_Players_Statistics`, `Team_Statistics`, and `Team_Iterator`.

Exercise: Browsing Ada Systems, cont.

5. Get the definition of the actual type declaration for `Total_Players_Statistics`.

Notice that the Environment displays the `Baseball` package specification with the type declaration highlighted.

6. Notice the structure of the type and the operations.

7. Get the definition of the next type in the package (`Team_Statistics`).

Notice that the Environment displays the private part of the package where the type is further defined.

Exercise: Browsing Ada Systems, cont.

8. Get the definition of the access type `(Team.Set)`.

The Environment displays the type in the `Set_Generic` package. The Environment has traversed a context clause and a link to get the definition of this package. Notice that this package is in a different world.

9. Return to `Baseball_Statistics`.

10. Move to the procedure call to `Data_Inputter.Get_Record` in the statement region.

11. Get the definition of the `Data_Inputter.Get_Record`.

Notice that the Environment now displays the `Data_Inputter` package specification with the declaration for the `Get_Record` procedure highlighted.

Exercise: Browsing Ada Systems, cont.

12. Get the definition of the body of

`Get_Record`.

Notice that you are now in the body of

`Data_Inputter`.

13. Return to the `Baseball_System` world.

14. Using the same methods as before, find the definitions of

— `Formatter.Print_Header`

— `Baseball.Percentage`

— `Baseball.Player_Iterator`

Seminar Outline

Basic Mechanisms

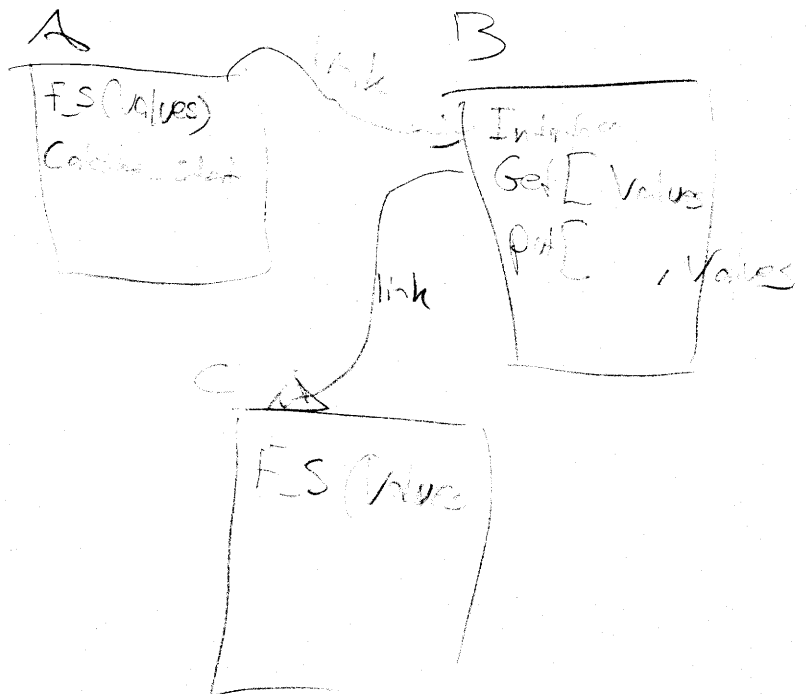
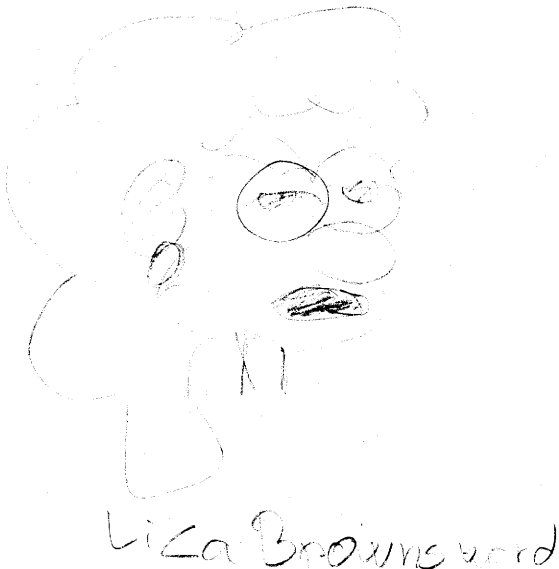
Ada Program Creation

Ada Program Modification

Simple Browsing

- Introduction to the Debugger
- Program Modification—Single-Unit Method
- Program Modification—Multiple-Unit Method

Additional Topics



Så som A og C's Values-type
er ens kan ville A ikke semantize
den B links til C.
OG A with use B.

Debugger Model

- Allows debugging of Ada programs at the Ada source level
- Acts as an outside agent to your program
 - Not compiled as part of your program
 - Used to control your program

Debugger Characteristics

- Operation is simply turning on; no recompilation is required
- Turning off the Debugger is not necessary
- One Debugger per session
- One job running with the Debugger at any time

Debugger Output

- Debugger displays sequential log in the Debugger window
 - Can scroll through the output
 - Can store Debugger output in a text object: `Text.Write_File`
 - Example

```
Break ("", 1, "");
Break at selected object.
The breakpoint has been created and activated:
Active Permanent Break 1 at .DEBUG_FACTORIAL.2S [any task]
Execute ("");
Break 1: .DEBUG_FACTORIAL.2s [Task : ROOT_TASK, #A60ED].
Execute ("");
Break 1: .DEBUG_FACTORIAL.2s [Task : ROOT_TASK, #A60ED].
Put ("");
Put selected object: %ROOT_TASK._1.THE_RESULT
```

2

Debugger window | text | JOB 250 STARTED 11 56 05 AM

Debugger Output, cont.

- Debugger displays current stopped location of the Ada unit
 - Uses standard Ada windows
 - User can use standard traversal mechanisms
 - Example

```
with Text_IO;
procedure Debug_Factorial (N : Natural) is
  The_Result : Natural := 1;
begin
  for I in 1..N loop
    The_Result := The_Result * I;
  end loop;
  Text_IO.Put_Line
    (Natural'Image (N) & "! is equal to " &
     Natural'Image (The_Result));
end Debug_Factorial;
```

```
= DEBUGGING DEBUG_FACTORIAL BODY v1111 ada ~ coded unit
```

Selection in Debugger

- Marks current location in stopped program
- Specifies the statement or declaration where a breakpoint should be placed
- Specifies the object or type to display

Basic Debugger Operations

- When the Debugger starts
 - Debugger window brought up to record all interactions
 - Starting new job message displayed in Debugger window

Basic Debugger Operations, cont.

- **Commands**

- Turn on the Debugger: ~~Meta~~ ~~Promote~~ *ESC - D(001)*
- Allow the program to run until breakpoints, exceptions, or normal program action occurs: Debug Execute
- Execute one step in the program: Debug Run
- Set a breakpoint at the selected statement or declaration: Debug Break
- Look at the selected value: Debug Put

Debug Run, Break, Place i de krapte de procedure

Script: Using the Debugger

Use the “Basic Debugger Operation” script to explore the following Debugger facilities:

1. Starting the Debugger.
2. Showing Debugger outputs.
3. Displaying the values of objects in the program.
4. Setting breakpoints.

Exercise: Debugging the Baseball Statistics Program

Use the *Rational Environment Basic Operations* and your notes to assist you. Execute and debug the `Baseball_Statistics` program to isolate an unhandled exception. The problem in the program will be fixed in an exercise in the next section.

1. Go to the `Baseball_System` world in your home world.
2. Create a Command window and execute the program `Baseball_Statistics`.

Exercise: Debugging the Baseball Statistics Program, cont.

3. Enter the following data (following each line with `Promote`):

```
Baker
4
2
1
Jones
0
0
0
```

Zero divide

An exception will occur. We need to use the Debugger to find where the exception is being raised.

4. Return to the Command window and reexecute the program with the Debugger. By default, the Debugger catches all exceptions at the point they are raised. This is what we need to find the problem.

Exercise: Debugging the Baseball Statistics Program, cont.

5. Execute the program in the Debugger:

`Debug Execute`

6. Enter the same data as before.
7. Note what statement was being executed when the exception was raised.

8. Display the values of the items in the numerical expression

`(The_Number_Hits / The_Times_At_Bat) :`

`Debug Put`

The exception is raised in the Percentage subprogram by an attempt to divide by zero. This problem will be fixed in an exercise in the next section of the course.

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

Simple Browsing

Introduction to the Debugger

- Program Modification—Single-Unit Method
- Program Modification—Multiple-Unit Method

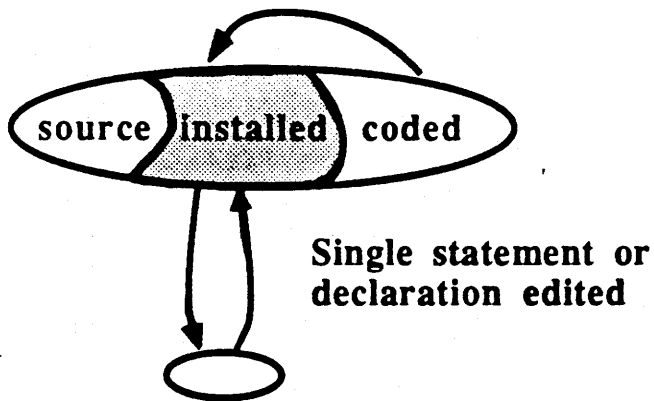
Additional Topics

Motivations

- Conventional model
 - Smallest unit that can be changed and must be recompiled is a compilation unit
 - Very small changes often require extensive recompilation
- Rational Environment model
 - Smallest unit that can be changed and must be recompiled is a statement or declaration
 - Very small changes require minimal recompilation

Basic Concepts

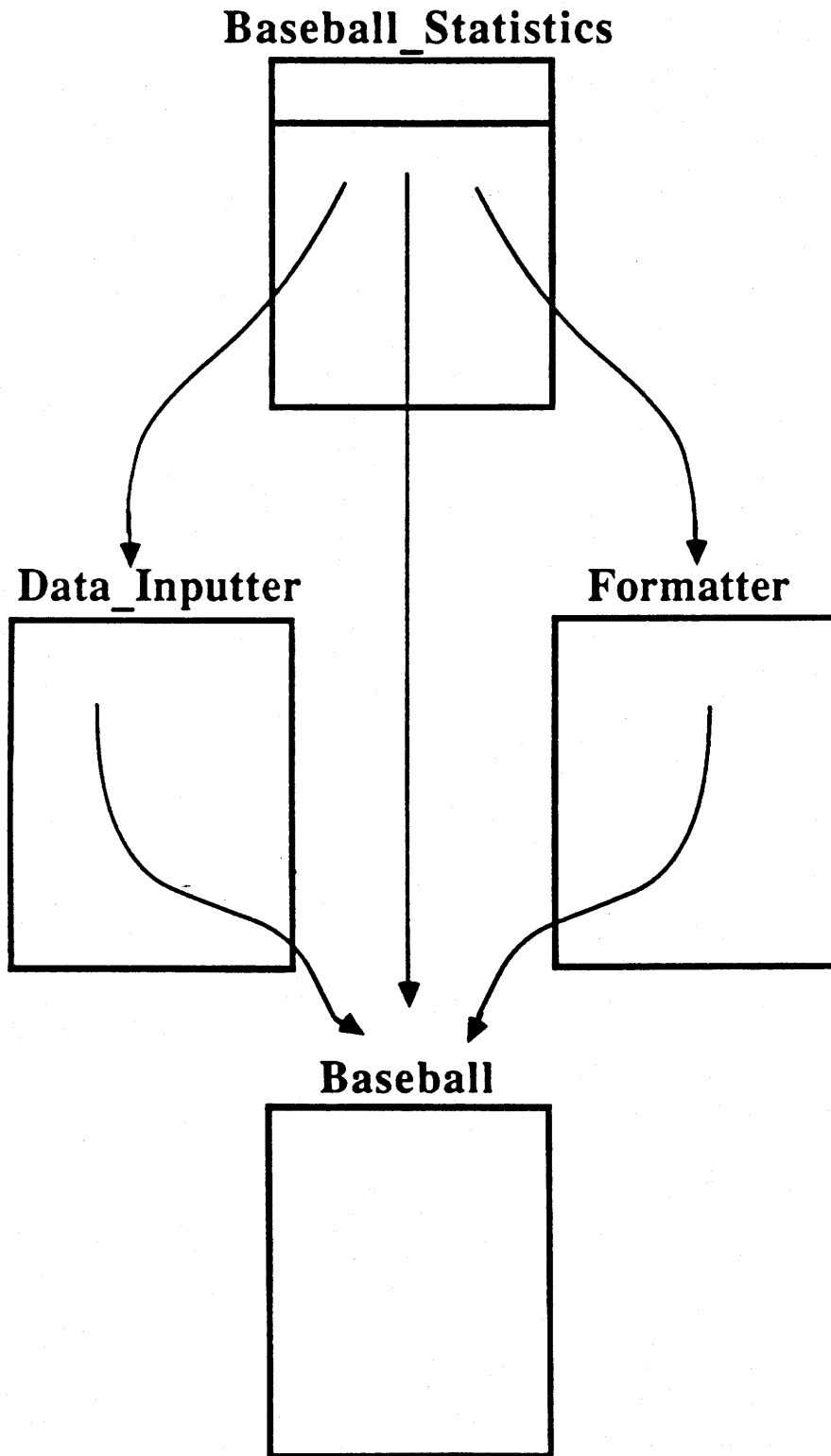
- Compilation dependencies on each unit and declaration are managed by the Environment
- Installed state allows incremental adding, changing, or deleting of statements or declarations that have no dependencies
- State transition



Basic Method

- Identify smallest element (statement or declaration) to be modified
- Demote compilation unit containing the element to installed state:
- Add, change, or delete element
 - Add element: - ; more than one element can be added at one time
 - Change selected element:
 - Delete selected element: -
- Format and semanticize as before
- Promote element when complete and no errors exist
- Recode system with automated compilation facility:
- Reexecute

Baseball Program



Baseball Program, cont.

```
with Set_Generic;
package Baseball is

  subtype Name is String;
  subtype Times_At_Bat is Natural;
  subtype Number_Hits is Natural;
  subtype Runs_Batted_In is Natural;
  subtype Percent is Float range 0.000..1.000;

  type Player_Statistics is
    record
      The_Name           : Name (1..20) := (others => ' ');
      The_Times_At_Bat   : Times_At_Bat := 0;
      The_Number_Hits    : Number_Hits  := 0;
      The_Runs_Batted_In : Runs_Batted_In := 0;
      The_Percentage     : Percent      := 0.0;
    end record;

  type Total_Players_Statistics is
    record
      Total_Times_At_Bat   : Times_At_Bat := 0;
      Total_Number_Hits    : Number_Hits  := 0;
      Total_Runs_Batted_In : Runs_Batted_In := 0;
      Total_Percentage     : Percent      := 0.0;
    end record;
```

Baseball Program, cont.

```

type Team_Statistics is private;
procedure Init_Team_Stats
    (The_Team_Stats : in out Team_Statistics);
procedure Add (The_Player_Stats : Player_Statistics;
    The_Team_Stats : Team_Statistics);
procedure Sum (Increment : in Player_Statistics;
    Summation : in out Total_Players_Statistics);
procedure Percentage
    (The_Player_Stats : in out Player_Statistics);
procedure Percentage
    (Total_Player_Stats : in out Total_Players_Statistics);

type Team_Iterator is limited private;
procedure Initialize (Iterator : in out Team_Iterator;
    The_Team_Stats : Team_Statistics);
function Value_Of (Iterator : Team_Iterator)
    return Player_Statistics;
procedure Get_Next (Iterator : in out Team_Iterator);
function Is_Done (Iterator : Team_Iterator) return Boolean;

private

package Team is new Set_Generic (Player_Statistics);
type Team_Statistics is access Team.Set;
type Team_Iterator is new Team.Iterator;

end Baseball;
```

Baseball Program, cont.

```
package body Baseball is
```

```
  procedure Init_Team_Stats  
    (The_Team_Stats : in out Team_Statistics) is  
  begin  
    The_Team_Stats := new Team.Set;  
  end Init_Team_Stats;
```

```
  procedure Add (The_Player_Stats : Player_Statistics;  
                The_Team_Stats : Team_Statistics) is  
  begin  
    Team.Add (S => The_Team_Stats.all, X => The_Player_Stats);  
  end Add;
```

```
  procedure Sum (Increment : in Player_Statistics;  
                Summation : in out Total_Players_Statistics) is  
  begin  
    Summation.Total_Times_At_Bat :=  
      Summation.Total_Times_At_Bat +  
      Increment.The_Times_At_Bat;  
    Summation.Total_Number_Hits :=  
      Summation.Total_Number_Hits +  
      Increment.The_Number_Hits;  
    Summation.Total_Runs_Batted_In :=  
      Summation.Total_Runs_Batted_In +  
      Increment.The_Runs_Batted_In;  
  end Sum;
```

Baseball Program, cont.

```
procedure Percentage
    (The_Player_Stats : in out Player_Statistics) is
begin
    The_Player_Stats.The_Percentage :=
        Float (The_Player_Stats.The_Number_Hits) /
        Float (The_Player_Stats.The_Times_At_Bat);
end Percentage;

procedure Percentage
    (Total_Player_Stats : in out Total_Players_Statistics) is
begin
    if Total_Player_Stats.Total_Times_At_Bat /= 0 then
        Total_Player_Stats.Total_Percentage :=
            Float (Total_Player_Stats.Total_Number_Hits) /
            Float (Total_Player_Stats.Total_Times_At_Bat);
    else
        Total_Player_Stats.Total_Percentage := 0.0;
    end if;
end Percentage;
```

Baseball Program, cont.

```
procedure Initialize (Iterator : in out Team_Iterator;
                    The_Team_Stats : Team_Statistics) is
begin
  Team.Init (Team.Iterator (Iterator), The_Team_Stats.all);
end Initialize;

function Value_Of (Iterator : Team_Iterator)
  return Player_Statistics is
begin
  return Team.Value (Team.Iterator (Iterator));
end Value_Of;

procedure Get_Next (Iterator : in out Team_Iterator) is
begin
  if not Is_Done (Iterator) then
    Team.Next (Team.Iterator (Iterator));
  end if;
end Get_Next;

function Is_Done (Iterator : Team_Iterator) return Boolean is
begin
  return Team.Done (Team.Iterator (Iterator));
end Is_Done;
end Baseball;
```

Baseball Program, cont.

```
with Baseball;  
package Data_Inputter is
```

```
  procedure Get_Record  
    (Value : in out Baseball.Player_Statistics);
```

```
  End_Of_Input : exception;
```

```
end Data_Inputter;
```

Procedure Get_Record (end of input)

Begin

T10.Number :=

T10.Number (Inputted of input)

T10.Get (End of input)

T10.Number

end

Baseball Program, cont.

```
with Text_Io;
package body Data_Inputter is

  package Tio renames Text_Io;
  package Nat_Io is new Tio.Integer_Io (Natural);

  procedure Put (Field_Name : String) is
  begin
    Tio.Put ("Enter the value for " & Field_Name & ": ");
  end Put;
```


Baseball Program, cont.

```

procedure Get_Record
    (Value : in out Baseball.Player_Statistics) is
    String_Length : Natural := 0;
    End_Of_Input_Mark : constant String (1..3) := "xxx";
begin
    loop
        begin
            Put ("name of player");
            Tio.Get_Line (Value.The_Name, String_Length);

            if Value.The_Name (1..String_Length) /=
                End_Of_Input_Mark then
                Put ("number of times at bat ");
                Nat_Io.Get (Value.The_Times_At_Bat);
                Tio.Skip_Line;

                Put ("number of hits ");
                Nat_Io.Get (Value.The_Number_Hits);
                Tio.Skip_Line;

                Put ("number of runs batted in ");
                Nat_Io.Get (Value.The_Runs_Batted_In);
                Tio.Skip_Line;
                Tio.New_Line;
                exit;
            else
                raise End_Of_Input;
            end if;
        exception
            when Tio.Data_Error =>
                Tio.Skip_Line;
                Tio.New_Line;
                Tio.Put_Line ("Invalid data. Try again.");
        end;
    end loop;

end Get_Record;
end Data_Inputter;

```

3- skip message (end of input)

Baseball Program, cont.

```
with Baseball;
package Formatter is

  procedure Print_Header;
  procedure Print_Player_Stats
    (Statistics : Baseball.Player_Statistics);
  procedure Print_Team_Stats
    (Statistics : Baseball.Total_Players_Statistics);

end Formatter;
```

Baseball Program, cont.

```

with Text_Io;
package body Formatter is

    package Tio renames Text_Io;

    package Nat_Io is new Tio.Integer_Io (Natural);
    package Flt_Io is new Tio.Float_Io (Baseball.Percent);

    procedure Put_Statistic_Values
        (At_Bat : Baseball.Times_At_Bat;
         Hits : Baseball.Number_Hits;
         Runs : Baseball.Runs_Batted_In;
         Percentage : Baseball.Percent) is
    begin
        Nat_Io.Put (At_Bat, 5);
        Nat_Io.Put (Hits, 5);
        Nat_Io.Put (Runs, 5);
        Tio.Put ("      ");
        Flt_Io.Put
            (Percentage, Fore => 0, Aft => 3, Exp => 0);
    end Put_Statistic_Values;

    procedure Print_Header is
    begin
        Tio.New_Line;
        Tio.Put_Line
            ("Name          " &
             "      ab      h      rbi      pct");
        Tio.Put_Line
            ("-----" &
             "-----");
    end Print_Header;

```

Baseball Program, cont.

```
procedure Print_Player_Stats
  (Statistics : Baseball.Player_Statistics) is
begin
  Tio.Put (String (Statistics.The_Name));
  Put_Statistic_Values (Statistics.The_Times_At_Bat,
    Statistics.The_Number_Hits,
    Statistics.The_Runs_Batted_In,
    Statistics.The_Percentage);

  Tio.New_Line;
end Print_Player_Stats;

procedure Print_Team_Stats
  (Statistics : Baseball.Total_Players_Statistics) is
begin
  Tio.Put ("Totals           ");
  Put_Statistic_Values (Statistics.Total_Times_At_Bat,
    Statistics.Total_Number_Hits,
    Statistics.Total_Runs_Batted_In,
    Statistics.Total_Percentage);

  Tio.New_Line;
end Print_Team_Stats;

end Formatter;
```

Baseball Program, cont.

```

with Baseball, Data_Inputter, Formatter;
procedure Baseball_Statistics is
  Team_Sums      : Baseball.Total_Players_Statistics;
  Team_Statistics : Baseball.Team_Statistics;
  Player_Iterator : Baseball.Team_Iterator;
begin
  Baseball.Init_Team_Stats (Team_Statistics);

  begin
    loop
      declare
        Current_Player : Baseball.Player_Statistics;

      begin
        Data_Inputter.Get_Record (Current_Player);
        Baseball.Percentage (Current_Player);
        Baseball.Sum (Current_Player, Team_Sums);
        Baseball.Add (Current_Player, Team_Statistics);
      end;
    end loop;
  exception
    when Data_Inputter.End_Of_Input =>
      null;
  end;

  Baseball.Percentage (Team_Sums);

  Formatter.Print_Header;
  Baseball.Initialize (Player_Iterator, Team_Statistics);

  while not Baseball.Is_Done (Player_Iterator) loop
    Formatter.Print_Player_Stats
      (Baseball.Value_Of (Player_Iterator));
    Baseball.Get_Next (Player_Iterator);
  end loop;

  Formatter.Print_Team_Stats (Team_Sums);

end Baseball_Statistics;

```

Exercise: Modifying Ada Programs

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Go to the `Baseball_System` in your home world.
2. Execute the `Baseball_Statistics` program.
3. Identify as many problems with the program as you can.

Zero divide
no exception needed
input based on old xxx data
program not.

Candidate - Yes

Exercise: Modifying Ada Programs, cont.

4. Notice the following problems, which will be corrected in the next series of exercises:
 - Program output does not allow adequate visual separation of the team total values from the last player's values.
 - Program output of the player's values does not line up with the headings.
 - A `Numeric_Error` exception is raised in `Baseball_Percentage`.
 - The user of the `Baseball_Statistics` program has no easy method to determine how to terminate input.

Script: Modifying Ada Programs— Adding Statements

Use the “Modifying Ada Programs: Adding Statements” script to assist you.

Modify `Print_Team_Stats` in the `Formatter` package of the `Baseball_Statistics` program to print out a dashed line *above* the team totals.

Exercise: Modifying Ada Programs— Adding Statements

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Modify `Print_Team_Stats` in the `Formatter` package of the `Baseball_Statistics` program to print out a dashed line *below* the team totals.
2. Reexecute the program to verify the change.

Script: Modifying Ada Programs— Changing Statements

Use the “Modifying Ada Programs: Changing Statements” script to assist you.

Modify `Put_Statistics_Values` in the `Formatter` package of the `Baseball_Statistics` program to change the field formatting parameter of the first `Nat_Io.Put` statement to align the output under the display headings.

Exercise: Modifying Ada Programs— Changing Statements

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, the previous script, and your notes to assist you.

1. Modify `Put_Statistics_Values` in the `Formatter` package of the `Baseball_Statistics` program to change the field formatting parameter of the second and third `Nat_IO.Put` statements to align the output under the display headings. (The field format value should be 8.)
2. Reexecute the program to verify the changes.

Exercise: Modifying Ada Programs— Changing a Subprogram

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, the previous script, and your notes to assist you.

1. Modify the first `Percentage` subprogram in the `Baseball` package of the `Baseball_Statistics` program to ensure that `Numeric_Error` does not occur because of an attempt to divide by zero.

Hint: Notice the second `Percentage` subprogram for a possible solution or create one of your own.

2. Reexecute the program to verify the changes.

Exercise: Modifying Ada Programs— Adding a Subprogram

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Add a procedure called `start_Message` to the `Data_Inputter` package to print out a message to the user before the first input value is requested.

Use the procedure provided below or one of your own.

```
procedure Start_Message is
begin
  Tio.New_Line;
  Tio.Put_Line
    ("Start of input for player statistics ...");
  Tio.Put_Line
    ("terminate with 'xxx' for player's name");
  Tio.New_Line;
end Start_Message;
```

Exercise: Modifying Ada Programs— Adding a Subprogram, cont.

2. Modify the main program, `Baseball-
_Statistics`, to print such a message by making an appropriate call to the new procedure.
3. Reexecute the program to verify the changes.

Exercise: Modifying Ada Programs— Changing Declarations

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

Modify the `Baseball_Statistics` program to change the data representation of the `Times_At_Bat` subtype in package `Baseball` to subtype `Times_At_Bat` is `Natural range 0..8` (note more constrained range).

1. Attempt to change the data representation in the `Baseball` package.
2. Stop when you are unable to edit the declaration because of an obsolescence message. The next section discusses the method for dealing with this message.

Units that are obsolesced by *Baseball*
Baseball/oper.
format/body

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

Simple Browsing

Introduction to the Debugger

Program Modification—Single-Unit Method

- Program Modification—Multiple-Unit Method

Additional Topics

Multiple-Unit Method

- Is necessary only when
 - A single-unit method fails because of an obsolescence message
 - Necessary changes are massive
- Uses an additional automated compilation feature
- Requires significantly more recompilation

Obsolescence Message

- Displays units obsolesced in a menu window
- Displays units obsolesced, not specific declarations or statements
- Allows basic traversal mechanism to be used to view any of the units listed

Automated Compilation Facility: Demote

- Demotes a selected Ada unit and any dependent units to the source state:

`Compilation Demote`

- Sends output logs to the standard output window
- Is similar in output format to `Compilation Make`

Multiple-Unit Method—Basic Steps

- Identify the minimum elements to be modified
- Demote the selected unit containing the elements and all dependent units to the source state: `Compilation Demote`
- Edit the necessary units: `Edit`
 - Incremental operations are not used
 - Arbitrary editing of source units
 - Format and semanticize as before
- Promote the system: `Compilation Make`
- Reexecute the system to verify any changes

Exercise: Making Changes across Multiple Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

Complete the modification to the `Baseball_Statistics` program. Change the data representation of the `Times_At_Bat` subtype in package `Baseball` to subtype `Times_At_Bat` is `Natural range 0..8`.

1. Use the automated compilation facility to demote the `Baseball` package and all its dependent units to the source state.
2. Change the data representation in the `Baseball` package. Use `semanticize` to be sure there are no errors.

*Define kernel symbols and/or use of
Total_Statistics, Total_Statistics refer to
exception after for inputs.
Use Total_Statistics > E. All showed!*

Exercise: Making Changes across Multiple Units, cont.

3. Use the automated compilation facility to return all units to the coded state.
4. Reexecute the system to verify the change.

Optional Exercise: More Changes across Multiple Units

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

Change the implementation of `Start_Message` in the `Data_Inputter` package so that it passes back a string that is the end-of-input mark.

The `Start_Message` procedure should prompt the user for this string rather than specify to the user what that string is. The `Get_Record` procedure should take this end-of-input mark as a parameter and use it to compare with user input for the player name to determine if input is complete. Thus both the specifications and bodies of both `Start_Message` and `Get_Record` need to be changed. Make the appropriate changes in `Baseball_Statistics` body to utilize these changes.

Review

- What do you need to do to use the Debugger?
- How do you find out more about additional Debugger operations?
- How do you move around Ada programs and when would you do this?
- How do you change Ada programs with the minimum recompilation?

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

Additional Topics

- Naming Conventions
- Library Objects Management
- Future Topics

Object Names

- Follow hierarchical directory structure
- Consist of two types
 - Absolute names *links*
 - Relative names

Absolute Names

- Are constructed from the root of the Environment (full pathnames)
 - `!Users.Pt_3.Experiment`
 - `!Io.Simple_Text_Io`

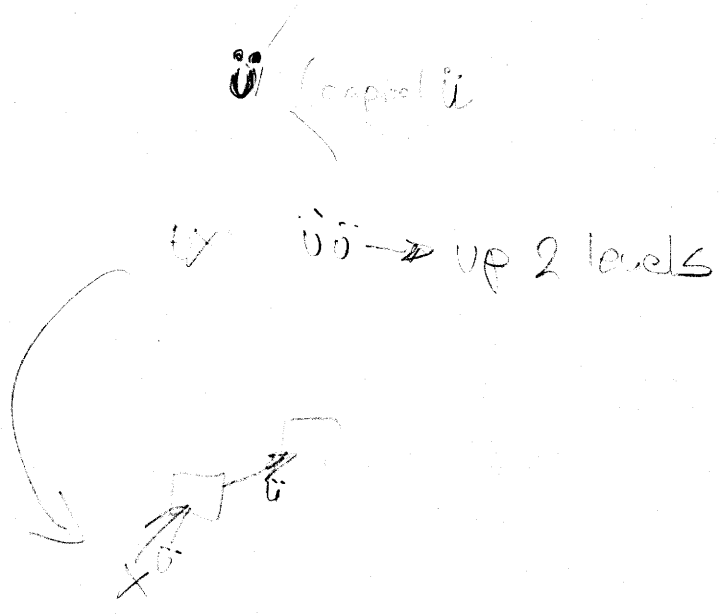
Relative Names

- Constructed from the current context
- Caret (^) means look in the parent object for the name
- Example

— Current context: !Users.Pt_3.Experiment

— Target: !Users.Pt_3.Debugging

— Relative name: ^Debugging



Wildcards in Names

- Provide a shorthand to reference objects
- Symbols *of course, etc. ?*
 - ^É means a string of characters of a simple name
 - ? means 0 or more ^É or .^É
- Examples from `Baseball_System` world
 - `BaseballÉ` references
 - `Baseball_System.Baseball,`
 - `Baseball_System.Baseball_Statistics`
 - ? references `Baseball_System` and all its units

Attribute Symbols

- Used to specify a restriction on names

- Ada units: 'spec OR 'body

— Examples: Foo'spec, Foo'body

- Versions: 'v

— Examples: Foo'V(1), Foo'V(1,3,5),

Foo'V(-2) *relative*

relative expressions

- Classes: 'c

— Examples: @'C(Ada), @'C(Ada, File),

@'C(Library)

Exercise: Naming

Use the *Rational Environment Basic Operations* and the *Rational Environment Keymap* to assist you.

Execute all steps of this exercise from a Command window using the `definition` command. Begin in any window.

1. Using the full pathname (absolute naming), go to your home world.
2. Using relative naming and attribute symbols, go to the specification of the `Baseball` package.
3. Using relative naming, go to `Experiment.Factorial`.

Exercise: Naming, cont.

4. Using relative naming, go to `Pt_n+1`, where `n` is your username.
5. Using either absolute or relative naming, return to your home world.

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

Additional Topics

Naming Conventions

- Library Objects Management

Future Topics

Versions

- Objects other than libraries can have multiple versions
- Each object has one current version and 0 or more deleted versions
- Deleted versions denoted by { }

Retention Count

- Specifies the number of deleted versions any object can have
- Default is 1
- Can be changed for an object:
`Library.Set_Retention_Count`

New Versions

- Creating text object versions: Enter, Promote,
 Object - X
- Creating Ada object versions: Enter, Promote,
 Install, Object - X

Library Information

- Controlling the library display

More info
 — To display more information about visible objects: `Explain Item` `Object - E`

tr
 — To change the display of the set of objects: `Object - !`, `Object - .`
version *full*

- Creating library listings

— To display object name, version, object class, updater's name, when last modified, size, and object status: `Verbose List`

— To display Ada units and their object state: `Ada List`

— To display only the information about files in library: `File List`

Workspace Management

- Libraries, Ada units, and files can be
 - Created
 - Copied or moved
 - Deleted or undeleted
 - Renamed
 - Frozen or unfrozen
 - Printed

Create Operations

- Commands are specific to the kind of library object
- Text objects
 - Create named text object: `Create Text`
- Ada objects
 - Create anonymous Ada unit: `Object` - `I`
- Library objects
 - Create named world: `Create World`
 - Create named directory: `Create Directory`

Copy and Move Operations

- Apply to any library object
- New Ada units are in the source state
- Commands
 - Copy/move a selected object to another library: `Object - C` OR `Object - M`
library, Ada or textfile ecc. *— Add unit or get down to source*
 - Copy/move a named object and change the object name: `Library.Copy` OR `Library.Move`

Delete Operations

- Apply to any library object
- Recoverable deletion
 - Delete selected object with no dependents or subordinate units: `Object - D`
 - Delete selected object and its dependents or subordinate units: `Compilation.Delete`
- Permanent deletion
 - Delete selected object and its dependents or subordinate units: `Compilation.Destroy`
 - Make recoverable deletions permanent: `Library.Expunge`

Undelete Operation

- Allows you to return to a specified version
- Applies to deleted files or Ada units
- Command

— Undelete a named object:

Library.Undelete

Rename Operation

- Applies to any library object
- Changes Ada objects to the source state
- Command

— Rename a named object: **Library.Rename**

with DO'S

① copy

② change name

③ destroy - original name

Freeze and Unfreeze Operations

- Frozen objects cannot be modified
- Any library object can be frozen
- Commands
 - Freeze selected object: `Library.Freeze`
 - Unfreeze selected object:
`Library.Unfreeze`

Printing Operations

- To print a selected object:
- To print a named object:

Rational prints ⇒ Object Print

Exercise: Managing Your Workspace

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Create a directory called `Documents` in your home world.
2. Create a file in your home world.
3. Move the new file into the `Documents` directory.
4. Change the name of the file.
5. Create a directory called `units` in your home world.
6. Copy the `Factorial` program into the `units` directory.

Exercise: Managing Your Workspace, cont.

7. Create a world called `Library_Experiment` in your home world.
8. Copy the `Documents` and `Units` directories into the new world.
9. Control the library display to determine the object state of the units in the `Baseball_System` world in your home world.
10. Delete the file in your home world.
11. Delete the `units` directory.
12. Delete the `Library_Experiment` world.

Exercise: Managing Versions

Use the *Rational Environment Basic Operations*, the *Rational Environment Keymap*, and your notes to assist you.

1. Set the retention count to four for `Baseball_System.Baseball_Statistics'body`.
2. Demote `Baseball_System.Baseball_Statistics'body` to the source state.
3. Set up the library display for `Baseball_System` to show all deleted objects.
4. Using the various commands that create versions in Ada objects, note the increasing version numbers in the banner of `Baseball_Statistics'body` and in the library display of `Baseball_System`.
5. Undelete one of the deleted versions.

Exercise: Managing Versions, cont.

6. Freeze the `Baseball_System` world. Note that the world and all of its objects are now frozen. Notice how this is indicated in the library display.
7. Try to add a text file to the `Baseball_System` world. (You should be unsuccessful.)
8. Try to delete the `Baseball_System` world itself. (This too should be unsuccessful. The error log of the delete operation will say that you can't delete frozen objects.)
9. Unfreeze the `Baseball_System`.
10. Add a text file to the `Baseball_System` world. (This should work.)

Review

- In what state are Ada units left after you copy them?
- Where do you find the commands that delete individual objects and entire libraries?
- Where do you find more information on naming in the Environment?

Seminar Outline

Basic Mechanisms

Ada Program Creation

Ada Program Modification

Additional Topics

- Naming Conventions

- Library Objects Management

- Future Topics

Nov - 2015

Future Topics

- **Work environment customization**

- Login procedures *ok*

- Key rebinding *ok*

- Switches

- Profiles and logs

→ controlling error reporting when/now

- **Environment utilities**

- Source management file utilities

- Source archive and restore

Future Topics, cont.

- Multiple library development
 - Searchlists
 - Links
 - Naming conventions
 - Additional Debugger facilities
 - Jobs and job scheduling

- Subsystems *- the way of developing the systems*

RATIONAL

Rational Environment Training

Scripts

Contents

Creating Ada Programs	1
Testing Ada Programs	7
Basic Debugger Operation	13
Modifying Ada Programs: Adding Statements	17
Modifying Ada Programs: Changing Statements	21

RATIONAL

Creating Ada Programs

Description

Introduces the steps in writing programs in libraries; introduces syntactic completion, semantic checking, and simple I/O; and explores moving the program between states.

The program prints a message in the standard output window using `Text_Io`. The program is built in a world in your home world.

Creating Ada Programs

Part 1. Steps 1 through 3 set up for program entry.

Step 1. Locate and go to your home world by pressing ~~Home~~. ^{ESC ↑}

Step 2. Locate and go to the Experiment world in your home world.

Step 3. Create a workspace for the program unit by pressing `Object` - `I`.

A new window appears with a `comp_unit` prompt in which to enter the program unit.

Notice the banner of the new window. Note the class of object being created, Ada, and its object state, source.


Part 2. Steps 4 through 13 enter the program and make it executable.

Step 4. Enter the following procedure declaration in the new window at the `comp_unit` prompt. Make sure the cursor is on the prompt, not adjacent to it.

```
procedure hello is
```

and complete the syntax of the subprogram fragment by pressing **Format**.

Notice that the `begin`, a statement prompt, and the `end Hello;` are automatically added. Also notice that capitalization has been changed and indentation automatically provided.

Step 5. Move to the statement prompt by pressing **ESC**  **Next Item**.

Step 6. Enter the following statement at the statement prompt. Again, make sure the cursor is on the prompt.

```
text_io.put_line("Hello World
```

and format by pressing **Format**.

Notice that the double quote, parenthesis, and semicolon are automatically added to the end of the statement.

Step 7. Check for semantic errors by pressing **Semanticise**.

Errors are indicated by underlines and a message displayed in the Message window.

Notice that a temporary name for the Ada unit has appeared in the Experiment world. The form of the name is `_Ada_#_`, where # is some number.

Step 8. More information about the errors is available by pressing **OBJECT**  **Explain Item**.

Additional error explanations are displayed in the Message window.

Step 9. Repair the error by adding the context clause to the program before the procedure reserved word. Move the cursor to the line that contains the procedure by pressing **Image** - **Begin DA**.



Creating Ada Programs

Step 10. Enter the context clause:

```
with text_io;  
  
and format.
```

Step 11. Again check for semantic errors by pressing .

Step 12. Promote the program to the installed state by pressing .

A message that the unit has been *installed* appears in the Message window. The banner indicates the name of the unit and the object state, installed. The banner also displays a running flag while the command is executing.

In the world, the temporary name is replaced by the actual subprogram name for the specification and body.

Step 13. Promote the program to the *coded* state by pressing .

A message that the unit has been coded appears in the Message window. The banner of the window displaying the Ada unit indicates the new object state.

Part 3. Steps 14 through 18 execute the program.

Step 14. Return to the Experiment world by pressing `Enclosing Object`.

Step 15. Open a Command window off the Experiment world by pressing `Create Command`.

Step 16. Enter the following statement in the Command window:

hello

Step 17. Execute the new program by pressing `Promote`.

The Environment links, loads, and elaborates all units of your program and then executes the program. Notice that the statement you typed is now reverse video and has become a *prompt*.

A new window, called the I/O window, appears on the terminal screen. Your message appears in this window.

The window is the standard input/output window used for the Standard_Input and Standard_Output files defined in the Text_Io packages. The banner of this window gives the job name and denotes it as a text object.

Step 18. You're done.

RATIONAL

Testing Ada Programs

Description

Introduces the use of Command windows for rapidly testing small programs.

The script uses the Hello program created in "Creating Ada Programs."

Testing Ada Programs

- Part 1.** Steps 1 through 6 set up a Command window for program entry.
- Step 1.** Locate your home world by pressing **Home**.
- Step 2.** Locate and go to the Experiment world in your home world.
- Step 3.** Create or return to a Command window by pressing **Create Command**.
- Step 4.** Expand the Command window by pressing **Window** - **I** twice to provide sufficient space to enter the test program.
- Step 5.** Go to the beginning of the Command window to see the entire contents by pressing **Image** - **Begin Of**.
- Step 6.** Move to the statement prompt by pressing **Next Item**.

Part 2. Steps 7 through 15 expand the Hello program to print the message repeatedly to illustrate how programs can be rapidly tested in Command windows.

Step 7. Enter the program name:

```
hello
```

and format.

Step 8. Place an outer loop statement around the program name by entering the following code before the subprogram call to Hello;. Move the cursor to the beginning of the line by pressing **Begin Of**.

Step 9. Enter the statements:

```
while count<5
count:=count+1;
```

and format.

Notice that the end loop and indentation are automatically supplied.

Step 10. Check for semantic errors by pressing **Semanticise**.

Errors are indicated by underlines and a message displayed in the Message window.

Step 11. More information about the error can be displayed in the Message window by pressing **Explain Item**. *OR ESC*

Notice that formatting, semanticizing, and error indication and handling are the same for Command windows as for Ada units in libraries.

Step 12. Repair the error by introducing an additional declaration before the begin reserved word. Move the cursor to that line and press **Begin Of**.

Step 13. Enter the declaration:

```
count:natural:=0;
```

and format.

Testing Ada Programs

- Step 14. Again check for semantic errors by pressing **Semanticise**. No errors should exist.
- Step 15. Reexecute the command procedure by pressing **Promote**. The Hello World message appears five times in the I/O window following the job name between lines of dashes to separate previous results.

Part 3. Steps 16 through 21 illustrate how test programs can be rapidly changed in Command windows.

Step 16. Return to the Command window by pressing .

Step 17. To change the loop count, turn the prompt off by pressing . *ctrl X*

This allows you to modify the text under the prompt.

The cursor should be on the line containing the `while...loop`.

Step 18. Change the 5 to 10 and .

Step 19. Again check for semantic errors by pressing .

No errors should exist.

Step 20. Reexecute the command procedure by pressing .

The Hello World message now appears ten times in the I/O window.

Step 21. You're done.

RATIONAL

Basic Debugger Operation

Description

Uses the Debugger to debug a simple program.

The program is the Factorial program. An existing version of that program has a bug in it. Although the bug is simple to find and might be obvious through observation, the Debugger will be used.

Basic Debugger Operation

Part 1. Steps 1 through 4 execute the program and discover its erroneous behavior.

Step 1. Locate and go to your home world.

Step 2. Locate and go to the Debugging directory.

Step 3. Create a Command window.

Step 4. Enter the following command and promote it:

```
debug_factorial(5 
```

The Environment creates an I/O window, if it does not already exist, and prints the answer, 16, in the window. The answer should not be 16 but 120.

Part 2. Steps 5 through 21 execute the program with the Debugger and isolate the program error.

Step 5. Return to the Command window that invoked the Debug_Factorial program.

Step 6. To invoke the Debugger with the program, press . ECC - RETURN

The Debugger window appears. The program has not begun execution.

Step 7. Run the program two steps by pressing twice.

This positions the Debugger to the start of your program. The Environment displays the program in a window where the statement or declaration to be executed next is highlighted (*selected*).

The Debugger is currently about to elaborate the first declaration of your program.

Step 8. Set a breakpoint at the second statement. Move the cursor to the second statement and press - repeatedly until the entire statement is selected.

Step 9. Create a breakpoint by pressing .

A message indicating the breakpoint number and location is displayed in the Debugger window.

This breakpoint allows you to interrogate the actions of the program each time through the loop.

Step 10. Execute the program by pressing .

The program stops at the breakpoint. A message indicating the breakpoint number and location is displayed in the Debugger window. The second statement in the program is still selected.

This is the first time through the loop. The program has not yet executed the selected statement.

Step 11. To display the value of I, select the occurrence of I in the *for* statement by moving the cursor to I and pressing - .

Basic Debugger Operation

Step 12. Display the value of the object by pressing **Debug Put**.

The value is displayed in the Debugger window. It should be 1, because this is the first time through the loop.

Step 13. To display the value of `The_Result`, select the occurrence of `The_Result`, which is on the left side of the assignment statement.

Step 14. Display the value of the object by pressing **Debug Put**.

The value is displayed in the Debugger window. It should also be 1. This is the initial value because the statement in the loop has not yet been executed.

Step 15. Execute the program again by pressing **Debug Execute**.

The program executes until the breakpoint at the second statement is reached. The window displaying the program unit has the statement selected.

Step 16. Display the value of `I`.

The value is displayed in the Debugger window. It should be 2, because this is the second time through the loop.

Step 17. Display the value of `The_Result`.

The value 2 for `The_Result` is displayed in the Debugger window.

Step 18. The Debugger shows that it will next execute statement 2. The program thus has executed this statement only once. The value of `The_Result` should be 1.

Notice that statement 2 sets `The_Result` to `The_Result plus I`. This is wrong. The correct algorithm should be to set `The_Result` to `The_Result times I`.

Step 19. You could now use the standard Environment facilities to modify the unit. You will learn how to use these facilities in the next section.

Step 20. You're done!

Modifying Ada Programs: Adding Statements

Description

Provides an example of how to make incremental changes to the algorithm of a subprogram in a package body. This example shows how to make such a change by *adding* statements to existing subprograms.

The program used to make the changes is called `Baseball_Statistics`. It is designed to calculate individual team batting statistics. It prompts for input about players (at bats, hits, runs batted in) and then calculates and displays batting percentages and team totals.

The program is built from these packages: `Baseball`, `Data_Inputter`, and `Formatter`. These are used in the main procedure called `Baseball_Statistics`.

The required change is in package `Formatter`. It is desired that the team totals, printed at the bottom of the output from the program, be separated by a line of dashes. We will add only the first dashed line in the script.

This type of change does not alter or remove any existing statements. It merely adds statements when printing the team totals. These changes demonstrate the incremental compilation capability of the Environment.

Modifying Ada Programs: Adding Statements

Part 1. Steps 1 through 3 find package `Formatter`, where the changes need to be made.

Step 1. Locate and go to your home world.

Step 2. Locate and go to the world called `Baseball_System` in your home world.

Step 3. Locate and go to the body of package `Formatter` in `Baseball_System`.

The body will be the second occurrence of `Formatter` in the `Baseball_System` world.

Part 2. Steps 4 through 10 make the first necessary change to the package.

Step 4. Demote the Formatter package body to the installed state by pressing **Install**.

The installed state allows incremental additions or changes without requiring the recompilation of other dependent units.

Step 5. Find the procedure `Print_Team_Stats` in the package.

Step 6. Move to the beginning of the first statement in the procedure.

Step 7. Create an insertion point for the new statement by pressing **Object** - **I**.

The Environment creates an insertion window in the top half of the window displaying `Formatter`. A temporary name is placed in the library under the body of `Formatter`.

Step 8. At the statement prompt in the new window, enter:

```
tio.put_line  
("-----" & "E")
```

and format.

(There are 30 dashes on each line.)

Step 9. Semanticize the statement.

This checks to make sure you will be able to add the statement to the program. There should be no errors.

Step 10. Promote the statement by pressing **Promote**.

Notice that the insertion window disappears and the new statement replaces the prompt in the subprogram. The temporary name is removed from the library.

Modifying Ada Programs: Adding Statements

Part 3. Steps 11 through 14 put the program back together again.

Step 11. Promote the body of `Formatter` to the coded state for execution by pressing `Promote`.

Step 12. Locate and go to the `Baseball_System` world by pressing `Enclosing Object`.

Step 13. Create a Command window, enter `Baseball_Statistics`, and execute the program to verify the effect of the changes by pressing `Promote`.

Step 14. You're done!

Modifying Ada Programs: Changing Statements

Description

Provides an example of how to change incrementally the algorithm of a subprogram in a package body. This example shows how to make such a modification by *changing* statements that already exist in subprograms.

The program used to make the changes is called `Baseball_Statistics`. It is designed to calculate individual team batting statistics. It prompts for input about players (at bats, hits, runs batted in) and then calculates and displays batting percentages and team totals.

The program is built from these packages: `Baseball`, `Data_Inputter`, and `Formatter`. These are used in the main procedure called `Baseball_Statistics`.

The required change is in package `Formatter` of the `Baseball` program. It is desired that the columns of numbers in the output be formatted with more space between the columns so that they align with the headers above them. We will fix only the first column in the script.

This type of change requires that existing I/O statements in package `Formatter` be changed.

Modifying Ada Programs: Changing Statements

Part 1. Step 1 finds the package that needs to be changed.

Step 1. Locate and go to the body of package `Formatter` in `Baseball_System`.

The body will be the second occurrence of `Formatter` in the `Baseball_System` world.

Part 2. Steps 2 through 7 make the first necessary change to the package.

Step 2. Demote the package body to the installed state by pressing .

The installed state allows incremental additions or changes without requiring the recompilation of other units.

Step 3. Find the `Put_Statistic_Values` procedure in the package. This is the procedure that must be changed.

Step 4. Locate and select the first statement in the procedure (the one with `At_Bat`) by moving the cursor to that line and pressing - repeatedly until the entire statement is selected.

Step 5. Edit that statement by pressing .

The Environment replaces that statement in the procedure with a statement prompt and creates a window in which to edit the statement.

Notice that the `Baseball_Statistics` library has a temporary name listed under the body of package `Formatter`.

Step 6. Change the value 5 to 8.

This is the field width for the value. It makes the column eight characters wide instead of five.

Step 7. Promote the statement by pressing .

The window disappears and the changed statement reappears in the procedure. The temporary name in the `Baseball Statistics` library is removed.

Modifying Ada Programs: Changing Statements

Part 3. Steps 8 through 10 put the program back together again.

Step 8. Promote the entire package body to the coded state by pressing **Promote**.

Step 9. Create a Command window, enter `Baseball_Statistics`, and execute the program to verify the changes by pressing **Promote**.

Step 10. You're done!