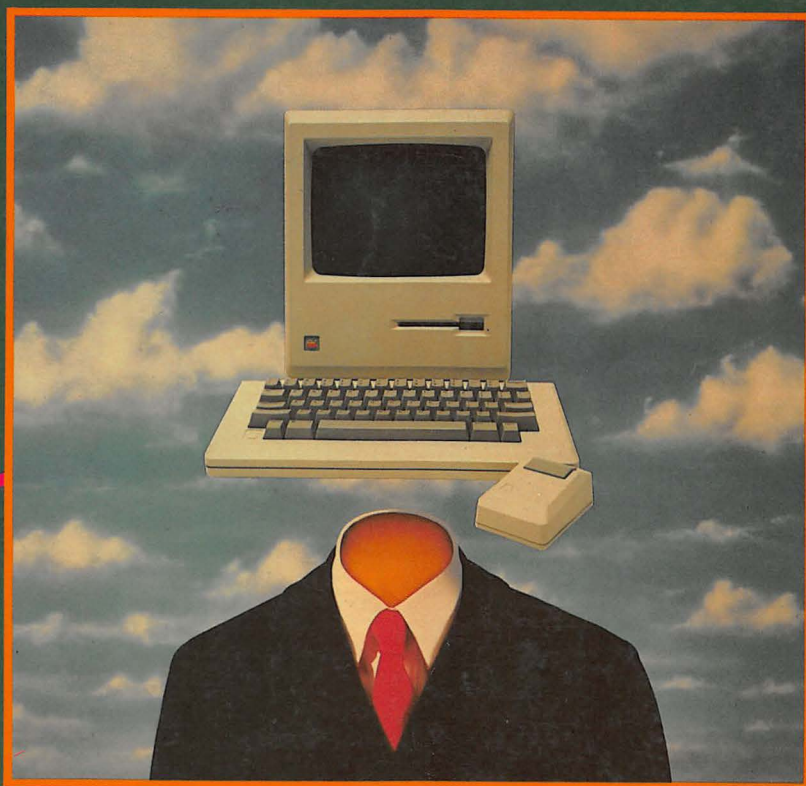


H A Y D E N

*Macintosh Library*

Apple  
Press™

# Introduction to Macintosh™ BASIC



S c o t K a m i n s

# **Introduction to Macintosh™ BASIC**

---

---



# **Introduction to Macintosh<sup>TM</sup> BASIC**

---

---

**Scot Kamins**

---



**HAYDEN BOOK COMPANY**

a division of Hayden Publishing Company, Inc.  
Hasbrouck Heights, New Jersey and Berkeley, California

Apple believes that good books are important to successful computing. The Apple Press imprint is your assurance that this book has been published with the support and encouragement of Apple Computer, Inc., and is the type of book we would be proud to publish ourselves.

Acquisitions Editor: Michael McGrath  
Production: The Compag Company, San Francisco  
Cover design: Jim Bernard  
Cover photo: Lou Odor  
Composition: Terry Robinson & Co., San Francisco  
Printed and bound by: Command Web Offset, Inc.

#### **Library of Congress Cataloging in Publication Data**

Kamins, Scot.

Introduction to Macintosh BASIC.

Includes index.

1. Macintosh (Computer)—Programming. 2. Basic

(Computer program language) I. Title. II. Title:

Introduction to Macintosh BASIC

QA76.8.M3K36 1984 001.64'2 84-15837

ISBN 0-8104-6550-7

**Copyright © 1984 by Hayden Book Company.** All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Macintosh is a trademark of Macintosh Laboratory, Inc., licensed to Apple Computer, Inc., neither of which is affiliated with Hayden Book Company. MacWrite, MacPaint, and Apple Press are trademarks of Apple Computer, Inc.

Printed in the United States of America

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>Printing</u>
84	85	86	87	88	89	90	91	92	Year

---

This book is dedicated to the joy of discovery and to all who support it, and especially to those who embody it, and most especially to

***John Dillon Riley***

Lumberjack Extraordinaire

who is, among those who support those who support the joy of discovery, the supportingest.



# Preface



---

**T**his is a nonacademic tutorial about Macintosh BASIC. It is primarily for people who have never programmed before. If that's you, and if you enjoy learning by *doing* rather than by *being talked at*, this is the book for you. It is based almost totally on your experiencing BASIC rather than being "taught" about it (as if any book could really teach a process). The book provides the framework, the hints, the suggestions, the experiences; you do the programming.

This tutorial assumes that you're sitting at your Macintosh with the book in front of you, ready to use the mouse and to type things on your computer's keyboard. It is not meant to be read on the bus going to or coming from work or school. It is a book to use, not to study.

This tutorial also assumes:

1. You have access to an Apple Macintosh computer and the Apple Macintosh BASIC language disk;
2. You have read the *Macintosh Owner's Guide* and know how to turn the computer on;
3. You want to learn how to program MacBASIC.

That's all you need. The book provides everything else. Except the pizza.

## **An Implied Contract**

Your using this book implies a contract between us. On the one hand, I agree to provide you with the best hands-on tutorial I know how to write: to provide good learning experiences, make timely suggestions, give you information that is true and relevant, stimulate you to teach yourself good programming practices, and keep you entertained so that your learning is fun. You, on the other hand, agree to follow the suggestions I make: to use this tutorial at the computer, to type in the programs, and to experiment as much and as often as you can with the Apple Macintosh BASIC language.

If I live up to my part of the bargain (and I have), and you live up to yours (I trust you), by the time you finish this book you'll know how to write a fairly complex BASIC program with (if you want) some fairly flashy graphics and even some animation. More importantly, you'll have learned sound programming practices that will serve you for as long as you pound a keyboard.

You won't learn all of the commands, statements, functions, tools, and other goodies that Apple's Macintosh BASIC provides; no single tutorial can cover them all effectively. But you'll learn a great deal, you'll learn it well, and you'll have a good time into the bargain.

SCOT KAMINS  
San Francisco, California

# Acknowledgments

---

---

---

---

**T**here are a number of people who either directly through their support or indirectly through their example contributed to this work.

First among these is **Donn Denman** who created Macintosh BASIC and who for the last two years has been my incredibly patient teacher.

**Chris Espinosa**, manager of Macintosh User Education, encouraged me to write this book in the first place.

**Steve Smith**, codeveloper of the Communitree Electronic Conferencing System, created the graphics for this book, corrected the manuscript, and found more program bugs than I care to admit.

**Mike McGrath**, West Coast Acquisitions editor for Hayden Books, gave constant reinforcement and provided excellent lunches.

**Ellen Romano**, astoundingly creative graphic artist at Apple Computer, played “naive user” and wrote a review of the manuscript that gave wonderful support just when I needed it.

**Dean Gengle**, author of *The Netweaver's Sourcebook*, said “Do the work” when I complained that I was stuck or bored or Just Couldn't Do It Anymore.

This is the work.



# Contents

---

---

---

Session 1	<b>Getting Started</b>	<b>1</b>
	<b>How This Book Is Organized</b>	<b>1</b>
	<b>How to Use This Book</b>	<b>3</b>
	<b>What's in What Session</b>	<b>3</b>
	<b>Conventions Used in This Book</b>	<b>6</b>
	<b>Now That You've Read This Far ...</b>	<b>6</b>
	<b>Your First Programming Tip</b>	<b>9</b>
Session 2	<b>The Programming Environment</b>	<b>11</b>
	How a Program Works	11
	Languages	12
	Syntax	12
	<b>The MacBASIC Environment</b>	<b>13</b>
	Entering Your First MacBASIC Program	14
	What "Run" Does	17
	A Note on Errors	18

---

	<b>Editing</b>	<b>18</b>
	Select, Copy, and Paste	<b>21</b>
	Cut	<b>21</b>
	Editing Letters, Words, and Phrases	<b>22</b>
	Clear and the Backspace Key: Removing Without Replacing	<b>25</b>
	Undo	<b>26</b>
	An Editing Shortcut	<b>27</b>
	Play Time	<b>28</b>
	<b>Summary</b>	<b>28</b>
<b>Session 3</b>	<b>Introduction to Variables</b>	<b>31</b>
	<b>Computer Arithmetic</b>	<b>32</b>
	Names of the Parts	<b>33</b>
	Meaningful Variable Names	<b>35</b>
	For Technotypes Only: How Variables Work	<b>36</b>
	<b>Making the Program More Flexible</b>	<b>37</b>
	Data vs. Information	<b>38</b>
	Punctuated PRINT	<b>38</b>
	<b>INPUT: Getting Information from the Outside World</b>	<b>41</b>
	INPUT Prompt Messages	<b>43</b>
	<b>Program Comments (!)</b>	<b>46</b>
	<b>Saving the Program</b>	<b>46</b>
	The Different Save Commands	<b>47</b>
	The Dialog Box	<b>48</b>
	All the Restrictions for Program Names	<b>49</b>
	<b>Summary</b>	<b>50</b>
<b>Session 4</b>	<b>Loops</b>	<b>53</b>
	Some Loop Definitions	<b>53</b>
	<b>The DO\LOOP Construct</b>	<b>54</b>
	A Loop Counter	<b>57</b>
	EXIT and IF... THEN: A Loop's Escape Hatch	<b>58</b>
	<b>IF ... THEN: A Powerful Construct</b>	<b>61</b>
	The Relational Operators	<b>61</b>
	IF... THEN with Other Keywords	<b>62</b>
	IF... THEN ... ELSE	<b>65</b>
	<b>About the Scroll Bar</b>	<b>66</b>
	<b>Summary</b>	<b>71</b>

Session 5	<b>Controlled Loops</b>	<b>73</b>
	<b>FOR\NEXT: More Control, Greater Complexity</b>	<b>74</b>
	How It Works	76
	An Implied STEP	77
	Using Variables in the FOR Line	78
	EXIT in a FOR\NEXT Loop	81
	Counter as Totaler	81
	Nesting: Loops Within Loops	82
	<b>First Taste of String Variables</b>	<b>86</b>
	Setting Up String Variables	86
	Section on Variable Redundancy Section	87
	INPUT with Strings	87
	How <i>Not</i> to Assign a String Value	88
	Identity Crisis: Numbers in Quotes	88
	Relational Operators with Strings	88
	A String Variable Idiosyncrasy: The Null String	90
	Experiment Time!	91
	<b>The Challenge: Program a BASIC Cash Register</b>	<b>92</b>
	Planning Phases	92
	The CLEARWINDOW Keyword	93
	The Challenge	93
	Don't Let the Bugs Get You Down	94
	<b>Summary</b>	<b>95</b>
Session 6	<b>Graphics and the Mouse</b>	<b>99</b>
	<b>The Graphics Area</b>	<b>99</b>
	<b>The PLOT Statement</b>	<b>100</b>
	Drawing Lines	102
	Combining PLOT Statements	104
	Quick Window Cleaning	104
	<b>From Points to Rectangles</b>	<b>106</b>
	Shapes Need Two Keywords	107
	Using Variables with Graphic Objects	107
	Slowing Things Down	110



<b>Some Mouse Words</b>	<b>111</b>
The Pointer Position	111
Finding the Column	111
About Out-of-Bounds Numbers	112
Finding the Row	115
Exceeding Vertical Boundaries	116
State of the Mouse Button	116
Play Time #1	118
For Those a Bit Lost	119
Play Time #2	120
The Save a Copy In . . . Command	121
More Control over Shape Placement	122
Why You Need to Inactivate IF MOUSEB	122
<b>Three Final Graphics Words</b>	<b>124</b>
<b>Summary</b>	<b>124</b>

## Session 7

<b>Random Subroutines</b>	<b>129</b>
---------------------------	------------

The RND Function	130
The INT Function	131
"Why Am I Doing This?"	133
RANDOMIZE for More Genuine Randomness	133
Integer Randomness	134
<b>Expressions</b>	<b>136</b>
Compacting Expressions	136
Precedence and Nested Expressions	137
Graphic Randomness	139
A Possible Solution	139
Random Bugs	140
A Few Final Modifications	142
<b>Subroutines—For Programs That are Easier to Read and Easier to Fix</b>	<b>143</b>
About that END PROGRAM . . .	147
<b>The Dice Game</b>	<b>149</b>
Game Rules	149
Planning the Program	149
Some Suggestions	150
<b>Summary</b>	<b>151</b>

<b>Session 8</b>	<b>Strings, Mostly</b>	<b>157</b>
	The LEN Function—Counting Characters	<b>157</b>
	<b>Substring Functions</b>	<b>158</b>
	And Just for the Exercise . . .	<b>163</b>
	Concatenation: Joining Strings Together	<b>164</b>
	<b>String Comparisons</b>	<b>165</b>
	ASCII: How One Word Can Be Less Than Another	<b>167</b>
	String Literals vs. String Variables	<b>169</b>
	Getting to ASCII: The Function ASC	<b>170</b>
	CHR\$—Producing Characters from ASCII Code	<b>172</b>
	Numbers	
	MacBASIC's ASCII Extensions	<b>174</b>
	Make Your Own ASCII Chart	<b>176</b>
	<b>TIME\$ and DATE\$</b>	<b>177</b>
	<b>Summary</b>	<b>180</b>
<b>Session 9</b>	<b>Arrays and Other Data</b>	<b>185</b>
	<b>What Arrays Look Like</b>	<b>185</b>
	DIM—Telling BASIC How to Set Up the Array	<b>187</b>
	Using Variables to Reference Array Elements	<b>191</b>
	Array Arithmetic	<b>192</b>
	Arrays for Strings	<b>195</b>
	Functions with String Arrays	<b>198</b>
	Multidimensional Arrays	<b>200</b>
	<b>READ and DATA—Another Way to Assign Values to Variables</b>	<b>204</b>
	The Data Pointer: Keeping Track of Used Items	<b>205</b>
	Moving the Data Pointer: The RESTORE Statement	<b>209</b>
	RESTORE at a Specific Location	<b>211</b>
	The READ\DATA\RESTORE Rules	<b>212</b>
	Cliffhanger	<b>213</b>
	<b>Summary</b>	<b>214</b>

Session 10	<b>Graphics Revisited</b>	<b>219</b>
	<b>PRINT and GPRINT</b>	<b>219</b>
	PRINT Destroys Graphics	<b>219</b>
	GPRINT: A Different Kind of PRINT	<b>221</b>
	What the Program's About	<b>222</b>
	<b>About Fonts</b>	<b>226</b>
	On Font Sizes	<b>230</b>
	Special Font Characters	<b>234</b>
	Time to Experiment	<b>236</b>
	<b>MacBASIC Painting Patterns</b>	<b>236</b>
	Thickening Points with PENSIZE	<b>238</b>
	On to Patterned Frames	<b>241</b>
	<b>ROUNDRECT: A Rectangle with Rounded Corners</b>	<b>243</b>
	<b>ASK: The Other Side of SET</b>	<b>247</b>
	Your Turn	<b>248</b>
	<b>Summary</b>	<b>249</b>
Session 11	<b>Advanced Decision Making</b>	<b>253</b>
	<b>Multiline IF . . . THEN\ELSE\ENDIF</b>	<b>254</b>
	<b>SELECT—The Complex Decision Maker</b>	<b>256</b>
	SELECT with Ranges	<b>258</b>
	Middle-of-Session Challenge	<b>261</b>
	<b>Boolean Values—How BASIC Makes Decisions</b>	<b>264</b>
	Some Hidden Boolean Operators	<b>266</b>
	The Booleans Data Type	<b>267</b>
	Booleans as Flags	<b>269</b>
	Keeping Boolean Values Straight	<b>269</b>
	MOUSEB~: A Boolean Keyword	<b>271</b>
	<b>Time to Solidify Learning</b>	<b>271</b>
	<b>Summary</b>	<b>272</b>



Session 12	<b>Program Planning</b>	<b>275</b>
	A Heartfelt Warning	276
	How to Do This Session	276
	<b>The First Step: Imagine the Outcome</b>	<b>277</b>
	What the Outcome Implies	277
	<b>Locking Down Some Goals: Program Specification</b>	<b>278</b>
	Overview	279
	Detailed Description	279
	Outlining the Specifications	284
	<b>Outline as Pseudocode</b>	<b>286</b>
	Pseudocode as Algorithm	287
	<b>Writing Code in Modules</b>	<b>287</b>
	Start with the Title Page	288
	Next: Drawing the Field	289
	Working Out the Timer Module	293
	Run Race: The Business End of the Program	294
	Adding the Racing Messages.Block	300
	The Test.For.Winner Module	302
	The Declare.Winner Block	302
	The Go.Again Module	304
	The End.Page Module	305
	Bringing the Modules Together	306
	The Module Groups: How They All Work	307
	Retrieving the Blocks	309
	The Initializing Routines	309
	Final Testing and Commenting	312
	Some Possible Enhancements	317
	<b>A Closing Comment</b>	<b>317</b>
	 <b>Appendixes</b>	
	<b>Appendix A</b> Useful Tables	<b>319</b>
	<b>Appendix B</b> Commands, Special Characters, Keywords, and Statements	<b>325</b>
	<b>Appendix C</b> Error Messages	<b>333</b>
	<b>Appendix D</b> Solutions to Bughouses	<b>339</b>
	 <b>Glossary</b>	<b>343</b>
	<b>Index</b>	<b>351</b>

# Figures and Tables

---

---

---

## Figures

---

<b>Figure 1-1</b>	Macintosh BASIC application icon	<b>8</b>
<b>Figure 1-2</b>	Welcome to Macintosh BASIC!	<b>8</b>
<b>Figure 2-1</b>	Untitled Listing Window ready for program	<b>13</b>
<b>Figure 2-2</b>	Your first Macintosh BASIC program	<b>16</b>
<b>Figure 2-3</b>	Result of Copy/Paste operation	<b>21</b>
<b>Figure 2-4a</b>	Selected text before replacement	<b>24</b>
<b>Figure 2-4b</b>	Result of replacing selected text	<b>24</b>
<b>Figure 2-5</b>	A Cut/Paste operation in progress	<b>25</b>
<b>Figure 3-1</b>	Simple addition program	<b>33</b>
<b>Figure 3-2</b>	Substituting meaningful variable names	<b>35</b>
<b>Figure 3-3</b>	Four arithmetic operations	<b>38</b>
<b>Figure 3-4</b>	Descriptive PRINT statements	<b>41</b>
<b>Figure 3-5</b>	Program modified to include INPUT lines	<b>43</b>
<b>Figure 3-6</b>	Commented listing of Minicalc	<b>45</b>
<b>Figure 3-7</b>	Save Text from File Menu	<b>47</b>
<b>Figure 3-8</b>	Dialog box for saving program	<b>48</b>

<b>Figure 4-1</b>	Inflation run amok	<b>55</b>
<b>Figure 4-2</b>	Error Message—"Loop without Do"	<b>56</b>
<b>Figure 4-3</b>	DO\LOOP with counter and IF... THEN EXIT	<b>60</b>
<b>Figure 4-4</b>	Program with relational operator	<b>63</b>
<b>Figure 4-5</b>	Ways to scroll contents of a window	<b>67</b>
<b>Figure 4-6</b>	Opening the Minicalc program from the File Menu	<b>70</b>
<b>Figure 5-1</b>	FOR\NEXT loop	<b>76</b>
<b>Figure 5-2</b>	FOR\NEXT loop using a STEP value of 3	<b>78</b>
<b>Figure 5-3</b>	Nested DO\LOOPS	<b>83</b>
<b>Figure 5-4</b>	Combination nested loops	<b>84</b>
<b>Figure 5-5</b>	Output of triple-nested loops	<b>85</b>
<b>Figure 5-6</b>	INPUT with string variable	<b>87</b>
<b>Figure 5-7</b>	Possible output of "The Challenge"	<b>94</b>
<b>Figure 5-8</b>	Listing of one solution to "The Challenge"	<b>97</b>
<b>Figure 6-1</b>	Simple PLOT program	<b>101</b>
<b>Figure 6-2</b>	Keeping the Pen down with a semicolon	<b>103</b>
<b>Figure 6-3</b>	Drawing more lines	<b>103</b>
<b>Figure 6-4</b>	Drawing a rectangle with FRAME RECT	<b>106</b>
<b>Figure 6-5</b>	Rectangles within rectangles within rectangles...	<b>110</b>
<b>Figure 6-6</b>	Local coordinates of output window	<b>114</b>
<b>Figure 6-7</b>	Dynamic graphics using FRAME RECT and mouse	<b>119</b>
<b>Figure 6-8</b>	Little Boxes	<b>121</b>
<b>Figure 6-9</b>	Little Boxes with BTNWAIT	<b>123</b>
<b>Figure 7-1</b>	RND chance	<b>131</b>
<b>Figure 7-2</b>	First and second run of RND function	<b>133</b>
<b>Figure 7-3</b>	First and second run of RND function with RANDOMIZE	<b>134</b>
<b>Figure 7-4</b>	Fifty random whole numbers	<b>135</b>
<b>Figure 7-5</b>	Random boxes	<b>140</b>
<b>Figure 7-6</b>	Random INVERTed OVALs	<b>143</b>
<b>Figure 7-7</b>	Teeny RND	<b>144</b>
<b>Figure 7-8</b>	Branching with GOSUB	<b>144</b>
<b>Figure 7-9</b>	"RETURN without GOSUB" error message	<b>148</b>
<b>Figure 7-10</b>	Answer to Pop Quiz	<b>153</b>

---

<b>Figure 8-1</b>	LEN, LEFT\$, RIGHT\$	<b>159</b>
<b>Figure 8-2</b>	MID\$	<b>160</b>
<b>Figure 8-3</b>	Result of character examination using MID\$(	<b>162</b>
<b>Figure 8-4</b>	Fabulous strings	<b>163</b>
<b>Figure 8-5</b>	String concatenation	<b>165</b>
<b>Figure 8-6</b>	"What's ASC about?"	<b>171</b>
<b>Figure 8-7</b>	CHR\$ alphabet	<b>173</b>
<b>Figure 8-8</b>	Getting characters from Key Caps	<b>174</b>
<b>Figure 8-9</b>	Pasting characters cut from Key Caps	<b>176</b>
<b>Figure 8-10</b>	TIME\$ and DATE\$	<b>177</b>
<b>Figure 8-11</b>	Summoning the Alarm Clock	<b>178</b>
<b>Figure 8-12</b>	Using the Alarm Clock to date-stamp a program	<b>180</b>
<b>Figure 8-13</b>	Code for the Bonus Question	<b>182</b>
<b>Figure 9-1</b>	Skeletal array structure	<b>186</b>
<b>Figure 9-2</b>	Output of First Array	<b>189</b>
<b>Figure 9-3</b>	Skeletal array structure showing value of elements	<b>190</b>
<b>Figure 9-4</b>	Using symbolic names for array elements	<b>192</b>
<b>Figure 9-5</b>	Array arithmetic	<b>194</b>
<b>Figure 9-6</b>	What to Find dialog box	<b>197</b>
<b>Figure 9-7</b>	Skeletal array for Sample\$	<b>197</b>
<b>Figure 9-8</b>	Two-dimensional array	<b>200</b>
<b>Figure 9-9</b>	Two-dimensional array with grades filled in	<b>202</b>
<b>Figure 9-10</b>	READ and DATA	<b>206</b>
<b>Figure 9-11</b>	Output of students' grades DATA	<b>206</b>
<b>Figure 9-12</b>	Gasping for DATA	<b>209</b>
<b>Figure 9-13</b>	Using RESTORE to reREAD DATA	<b>210</b>
<b>Figure 9-14</b>	How RESTORE works with labels	<b>212</b>

<b>Figure 10-1a</b>	Rectangle before execution of PRINT statement	<b>220</b>
<b>Figure 10-1b</b>	Damage to graphics by PRINT statement	<b>220</b>
<b>Figure 10-2</b>	First output of GPRINT	<b>222</b>
<b>Figure 10-3</b>	Close-up of text and GPRINT output	<b>223</b>
<b>Figure 10-4</b>	Selecting area of text/GPRINT for a closer look	<b>224</b>
<b>Figure 10-5</b>	Using the Hand to move Fat Bits picture	<b>226</b>
<b>Figure 10-6</b>	All of the fonts	<b>227</b>
<b>Figure 10-7</b>	Really strange lines	<b>228</b>
<b>Figure 10-8</b>	Even stranger line	<b>229</b>
<b>Figure 10-9</b>	Q's	<b>231</b>
<b>Figure 10-10</b>	Historic quote	<b>232</b>
<b>Figure 10-11</b>	Pattern 11	<b>237</b>
<b>Figure 10-12</b>	All 38 patterns	<b>237</b>
<b>Figure 10-13</b>	Thick line	<b>239</b>
<b>Figure 10-14</b>	Thick FRAME RECT	<b>240</b>
<b>Figure 10-15</b>	Thick and thin FRAME OVALs	<b>241</b>
<b>Figure 10-16</b>	Robot in jail	<b>242</b>
<b>Figure 10-17</b>	RECT vs. ROUNDRECT	<b>243</b>
<b>Figure 10-18</b>	ROUNDRECT within RECT	<b>244</b>
<b>Figure 10-19</b>	Different roundness factors	<b>245</b>
<b>Figure 10-20</b>	Concentric INVERTed ROUNDRECTS with robot	<b>246</b>
<b>Figure 10-21</b>	ASKing about SET options	<b>247</b>
<b>Figure 11-1a</b>	Oval branch of Multiline IF... THEN\ELSE\ENDIF	<b>255</b>
<b>Figure 11-1b</b>	Rectangle branch of Multiline IF... THEN\ELSE\ENDIF	<b>255</b>
<b>Figure 11-2</b>	"Couldn't find a Case that Matched" error statement	<b>258</b>
<b>Figure 11-3</b>	Output of SELECT structure	<b>261</b>
<b>Figure 11-4a</b>	Using SELECT to get font characteristics and sample	<b>263</b>
<b>Figure 11-4b</b>	Characteristics and sample of Font 11	<b>264</b>
<b>Figure 11-5</b>	Booleans	<b>266</b>
<b>Figure 11-6</b>	NOT and NOT NOT	<b>270</b>

<b>Figure 12-1</b>	Ending of <i>The Great American Sheep Race</i>	<b>278</b>
<b>Figure 12-2</b>	Title page	<b>280</b>
<b>Figure 12-3</b>	The field	<b>280</b>
<b>Figure 12-4</b>	Start of the race	<b>281</b>
<b>Figure 12-5</b>	Leader of the pack	<b>282</b>
<b>Figure 12-6</b>	Crossing the finish line	<b>282</b>
<b>Figure 12-7</b>	The champion sheep	<b>283</b>
<b>Figure 12-8</b>	Good-Baa-ye page	<b>284</b>
<b>Figure 12-9</b>	Sheep droppings	<b>298</b>
<b>Figure 12-10</b>	Close-up of sheep	<b>299</b>
<b>Figure 12-11</b>	Rectangle defined by 0, 0; 200, 100; and dot at 200, 100	<b>300</b>

## Tables

<b>Table 3-1</b>	Common Arithmetic Operators	<b>32</b>
<b>Table 3-2</b>	Simulated Variable Names	<b>36</b>
<b>Table 4-1</b>	Relational Operators	<b>62</b>
<b>Table 5-1</b>	Variables Used to Hold Running Totals	<b>82</b>
<b>Table 5-2</b>	Non-Matching Strings	<b>89</b>
<b>Table 7-1</b>	Rules of Precedence	<b>137</b>
<b>Table 8-1</b>	ASCII Character Chart	<b>168</b>
<b>Table 10-1</b>	Available Fonts	<b>233</b>
<b>Table 10-2</b>	Special Characters	<b>234</b>
<b>Table 10-3</b>	Font 11	<b>235</b>

# SESSION

## 1

## Getting Started

---

**T**his book is a step-by-step, hands-on, experience-based, hyphen-filled tutorial introduction to Apple's Macintosh BASIC. This session is the only one in which you won't do much more than read. Its purpose is to provide you with a frame of reference about the tutorial; specifically, it tells you how the book is organized, what each session generally covers, and how to use the tutorial. If you don't care about such stuff and want to get right into the fun of learning how to program, go immediately to the end of this session and read the section called "Now That You've Read This Far..."; you can read the rest of the session later on, when somebody else is using the Macintosh.

### How This Book Is Organized

---

This tutorial has 12 sessions. All but this one are *experiential* in nature—that is, you spend a great deal more time doing than reading. Each session is built on the previous one; Session 3 makes use of what you learned in Session 2, Session 4 uses everything you learned in Sessions 2 and 3, and so on. The examples in later sessions are more complex than the ones in earlier sessions, and the programs you're asked to write on your own will get more and more complex as you move further along.

Most sessions follow the same format. Each one begins with a brief introduction saying what that particular session is about. Then you're immediately asked to do something new; if it's at all complex, you get step-by-step instructions. This learning experience is followed by an explanation of what you did and why it's good to be able to do that sort of thing. You're continually encouraged to experiment on your own. Each session continues with more "guided experimenting," using new BASIC instructions and Macintosh BASIC's many built-in programming tools. Comments are thrown in here and there to give you hints about ways to improve your programming style, and an occasional "pop quiz" lets you see how you're doing. At points where things might go wrong, you'll often find suggestions about what might have caused the problem and how to fix it.

Sessions 2 through 11 have summaries at the end. The summaries include a list of the new terms used in the session; abbreviated descriptions of the commands, statements, and programming symbols you just learned; and answers to the pop quizzes. There's also a summary list of every command, menu item, and keyword-programming statement or symbol you've learned so far. Finally, just so you can prove to yourself that you've learned something, each of these sessions ends with a section called "Bughouse." A Bughouse is a short program or program fragment that should work, but doesn't. Your job is to make it work by finding and correcting the mistakes or bugs. Bughouse solutions appear in Appendix D.

At the back of the book, just before the Glossary and Index, are several appendixes; they'll come in handy after you've finished the book and need a quick source of information. Appendix A is a collection of all the useful tables in the book, including something called the ASCII chart, the meaning of which becomes clear only after you've had some programming experience (sorry). Appendix B is a summary of all the commands and statements the book covers. Appendix C gives the error messages you're likely to get, lists the reasons the error might have happened, and makes suggestions for fixing the bug. Appendix D holds my solutions to the Bughouse problems that appear at the end of most of the sessions.



## How to Use This Book

---

No two people learn in the same way. Some people like to be guided every step of the way; others can't stand any kind of directions and like to experiment completely on their own. Most people, not surprisingly, are somewhere in the middle. Here's a list of different ways you can use this book, if you're looking for suggestions:

*If you're brand-new to programming*, begin with Session 2 and go straight through to the end. Do every suggested exercise, write every recommended program, do all the Bughouses and experiment every chance you get. There are suggestions for experimentation in all the sessions.

*If you've already started learning Macintosh BASIC on your own from the reference manual* and you like learning that way, use this book as a supplement. If you have trouble figuring out how to use one of MacBASIC's statements, look it up in this book and do the exercises and experiments that are related to it.

*If you've been programming in other BASICs*, read "What's in What Session" (below), Appendix B (the summary of MacBASIC commands and statements), the Glossary, and all the material at the end of Sessions 2 through 11. That way you'll get a feel for the unique features of Macintosh BASIC (there are quite a few) and where you'll have to start in the book to learn them all.

*If you're an experienced programmer in a structured language*, follow the course I've just recommended for people experienced in other BASICs. You'll be amazed at the control structures that Macintosh BASIC offers. This book, for example, doesn't teach the GOTO statement at all. Macintosh BASIC has such sophisticated control structures that you'll never be subjected to the indignity of using GOTO or POP.

## What's in What Session

---

It's difficult to condense into a single paragraph what you'll learn in each session; learning to program is a process that isn't nearly as quantifiable as what the computer itself produces. However, the following brief summaries will give you an idea of what each session covers.

Session 2, “The Programming Environment,” introduces you to some fundamental programming concepts and gives you experience in using a few of MacBASIC’s programming tools. You’ll learn about windows in BASIC, including what happens where on the screen. You’ll enter and change your first program, and in the process you’ll learn how to use the Mac’s Editor.

Session 3, “Introduction to Variables,” deals with the way computers manipulate information. You’ll learn about variables, the places in the computer’s memory where BASIC puts the data you give it. Further sections show you how to think like a computer (since they’re too dumb to think like people) and how you can use that knowledge to write programs with few errors. In the process you’ll learn more “environment” commands (like the ones that let you store the programs you’ll write) and new BASIC statements—including the all-important INPUT statement, which lets your program interact with you and other people.

Session 4, “Loops,” teaches you how to make BASIC double back on a program path (yes, programs provide pathways) and repeat a series of steps. You’ll learn why loops are important and how to use them. You’ll get experience using the powerful IF...THEN and DO\LOOP constructs, and you’ll have your first pop quiz so you can see how you’re doing.

Session 5, “Controlled Loops,” gives you advanced practice using more complex programming structures. You’ll learn about the FOR\NEXT loop and how to combine different kinds of loops to get your programs to do more work with fewer instructions. You’ll also get your first experience with string variables, which let you do all kinds of interesting things with words and special characters.

Session 6, “Graphics and the Mouse,” gets you into the special world of Macintosh graphics. You’ll learn how the Mac’s graphics screen is set up, and you’ll immediately start writing graphics programs. You’ll find out how to plot points and draw rectangles and ovals. Finally, you’ll write some strange little sample programs that let you do unspeakable things on the screen with the mouse.

Session 7, “Random Subroutines,” introduces the use of subroutines, one of the most important concepts in programming. You’ll also learn about random numbers, a useful tool for games

and simulations, by writing and using subroutines that generate and use them. Before the session is over, you'll have written a full-fledged game for fun (and possibly profit).

Session 8, "Strings, Mostly," offers programming tools for manipulating words. You'll learn about an international computer coding system called ASCII, and you'll see how to use some of the Mac's *Desk Accessory* programs to help you "code" more efficiently.

Session 9, "Arrays and Other DATA," covers some advanced techniques for setting up and using variables. You'll learn more about using MacBASIC's powerful programming tools (including ways to make sweeping program changes with a few keystrokes) and how to write some very useful programs for storing information.

Session 10, "Graphics Revisited," brings you more advanced techniques for manipulating the Mac's incredible graphics tools. You'll find out how to mix text and graphics together and how to produce text in a variety of type styles and sizes. Finally, you'll get practice filling in the shapes you draw with any of 38 different patterns.

Session 11, "Advanced Decision Making," shows you how to use two new sets of statements—one called Multiline IF and one called SELECT CASE—to let your programs make complex decisions. You'll also learn about booleans, a special kind of variable that can deal only with the concepts *true* and *false*.

Session 12, "Program Planning," is the last session in the book. It makes explicit what's been implicit throughout the tutorial—that good planning is essential to good programming. You'll learn two special planning techniques, *pseudocode development* and *top-down programming*, and then put what you learn into immediate practice by developing and writing what might be the most important program of your coding career—the fabled graphics program, *The Great American Sheep Race*.

---

## Conventions Used in This Book

---



Important rules and principles are summarized and emphasized by being printed in the margin in color like this.



Certain features included in this book are designed to make your learning of MacBASIC as painless, but efficient, as possible. Strewn throughout each session are exercises that give you hands-on experience in MacBASIC. The exercises are headlined “Do This” and are further identified in the left margin by a small drawing like the one shown on the left.

Incidental but still important pieces of information will be brought to your attention throughout the sessions by the heading “For Your Information” and a small drawing in the left margin like the one at left.

Last, whenever a pop quiz is sprung on you so you can see how you’re doing, you can get some advance warning by looking for the marginal drawing like the one shown at left.

---

## Now That You’ve Read This Far . . .

---

You might as well start getting your money’s worth out of this book. If you’ve already got BASIC up and running, skip to the section called “Your First Programming Tip.” If not . . .



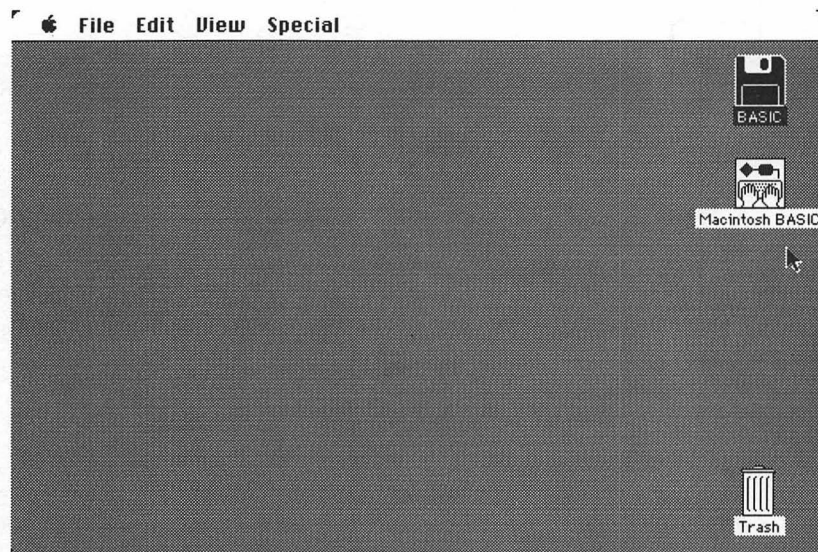
## Do This

To **boot** means to get something started. Here's how to boot Macintosh BASIC:

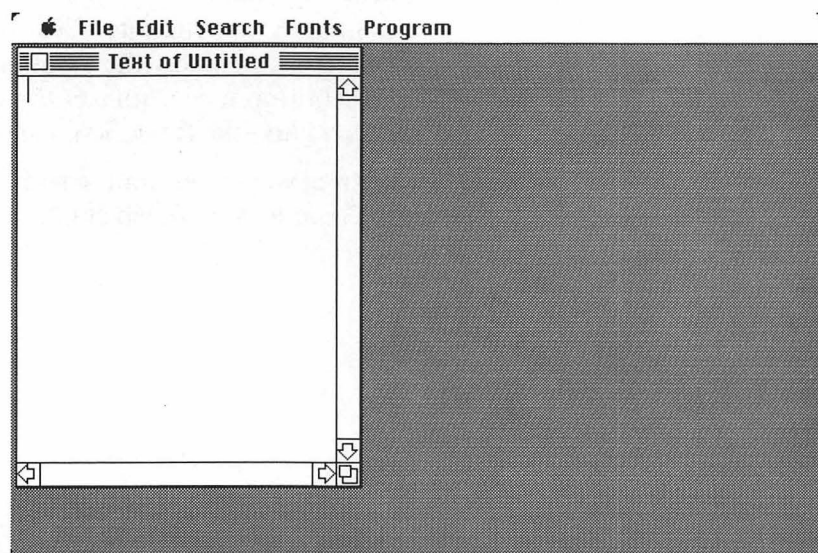
1. Insert the BASIC disk into the drive.
  - a. Hold the disk so that the round metal piece in the center is facing down and the metal clip on the end of the disk is facing away from you.
  - b. Slide the disk all the way in until the Mac grabs it.
2. Find the Macintosh BASIC application icon (see Figure 1-1).
3. Roll the mouse so that the pointer is on top of the icon and press the mouse button twice, quickly. If you can't find the BASIC icon, roll the mouse until the pointer is over the icon that looks like a disk (it's probably in the upper right corner) and press the mouse button twice, quickly. A window will "open" and show the names and icons for the documents (files) on the disk. Locate the BASIC icon, position the pointer on it, and press the mouse button twice, quickly. If no window opens or you still can't find the BASIC icon, you've got the wrong disk.

If everything went well, your screen should look like Figure 1-2. Welcome to Macintosh BASIC!





**Figure 1-1** Macintosh BASIC application icon



**Figure 1-2** Welcome to Macintosh BASIC!

## Your First Programming Tip

---

This tip will serve you better than any other you'll find in this book: *experiment your brains out*. Try everything. When you're feeling bored, write totally strange programs. Where the book says not to do something, try doing it anyway. Do what the book suggests, then do the opposite and see what happens.

While no two people learn in the same way, the only way that anybody can learn to program is—to program. Nobody ever learned to program just by reading about it. And nobody has ever developed creativity just by listening to other people (including writers) talking about their own creative experiences. Start now! All you have to do is turn the page. . . .



## SESSION



# The Programming Environment

---

**M**acintosh BASIC is more than just a programming language. It's a complete programming system that includes the language itself, the Macintosh Editor for entering and changing program codes, Macintosh accessories to help program development and to provide a little diversion from the rigors of programming, and a series of special menu items for testing your programs and getting the bugs out (that is, correcting any errors).

In this session you'll become familiar with the Macintosh BASIC programming environment, particularly the Editor. You'll learn what a program is, how it works, and how computer languages are like human languages. You'll get practice using the mouse, and you'll write your first BASIC program. You'll also become more familiar with a lot of the terms you'll come across when you use MacBASIC or any other Macintosh application.

### How a Program Works

Computers are inherently stupid; they rely totally on programs to tell them what to do. No program, no action. All a computer can do is follow instructions.

A computer does exactly and only what a program tells it to do; it can do nothing on its own.

A **program** is a set of coded instructions that makes a computer perform a task or series of tasks. When you tell your Mac to execute a program, it carries out the program's instructions with absolute precision and in a rigid order.

BASIC looks at the first line of **code**, or computer-language instruction, and does what it says to do. When BASIC finishes that instruction, it goes on to the instruction immediately following (unless the first instruction said to go to some other part of the program) and does what that code says to do. This goes on, line by line, until BASIC runs out of instructions to follow and the program ends.

### Languages

Microcomputers can't yet understand natural languages, the languages that people speak. The native language of microcomputers is binary—a numeric system based on the two numbers 0 and 1. In fact, the only thing micros really understand is 0's and 1's (I told you they were stupid). In order to do even the simplest task (say, add two numbers together), the computer must process dozens of 0–1 combinations.

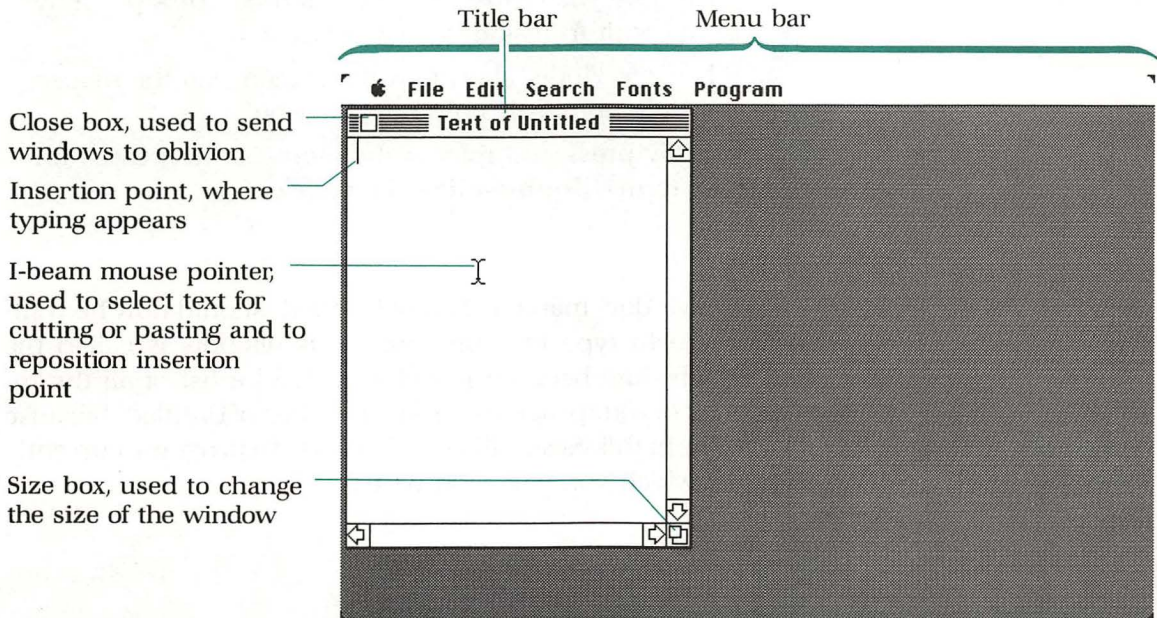
Macintosh BASIC is a higher-level language—that is, it is closer to natural language than to the computer's language. Higher-level languages allow you to avoid the tedium of typing in endless 0's and 1's. You can plan a program in human language, translate it into BASIC, and then type it into your Mac. The BASIC interpreter (yet another program, one that lives in your Mac) further translates your program into a form the computer can understand. You don't have to think about the interpreter; it does its job automatically.

### Syntax

Like any language, MacBASIC has a grammar and a vocabulary. The grammar is formal: the rules of its syntax are fairly strict, you've got to be careful about punctuation, and spelling counts. There aren't any grades, though; the worst thing that can happen if you make a mistake is that your program won't work right. But more on that later when you read about error correction, or **debugging**.

## The MacBASIC Environment

You write Macintosh BASIC programs in an environment called the BASIC **shell**. When the shell is on your screen, the screen should look like Figure 2-1.



**Figure 2-1** Untitled Listing Window ready for program



### If Something Goes Wrong

You can't see the shell—you haven't selected BASIC from the finder.

#### To Fix It

1. Make sure you've inserted the MacBASIC disk properly in the built-in disk drive (see Session 1).
2. When the Finder shows on the screen, roll the mouse until the pointer is on the BASIC icon.
3. Quickly press and release the mouse button twice (in Mac terms, **double-click** the mouse).

The window marked "Text of Untitled" should now be waiting for you to type in a program. This window is called the **listing window** because it will soon hold a list of all the instructions in your program. It's labeled "Text of Untitled" because it holds (or, in this case, will hold) the **text** of a program, currently untitled, which you will soon write.

Text is any collection of characters.



### For Your Information

**All New Programs Untitled** All listing windows whose programs have never been stored start with the name *Untitled*. Later on, when you store the program, BASIC will ask you for a more appropriate name.

### Entering Your First MacBASIC Program

The Big Moment has arrived: It's time for you to type in your first program. This program just puts some words on the screen so you can practice using the Editor. Don't be nervous; you can't hurt the computer while you program, unless you jam the keyboard into the computer display (considered by most experts to be a rather inelegant debugging technique).

The program you're about to write has just a single line of BASIC code. The line consists of the keyword **PRINT**, which tells the computer to show some text on the screen, and some words enclosed in quotes (technically called a **quoted string** because it's a string of characters enclosed between quote marks). A **keyword** is any word or phrase that has a specific meaning to the computer.



The Return key's over on the right-hand side of the keyboard.

To drag is to roll the mouse while holding down the mouse button.

### Do This

Type the following line into your computer, just as it appears, except type your own name instead of mine. Then press the **Return** key.

```
PRINT "Hello, Scot. What's happening?"
```

Now choose Run from the Program menu.

1. Roll the mouse until the **pointer** is over the word Program in the **menu bar**.
2. Press and hold down the mouse button.
3. Drag the pointer to the word Run.
4. Release the mouse button.

Something like Figure 2-2 should appear on your screen.

### If Something Goes Wrong

MacBASIC might display one of these error messages:

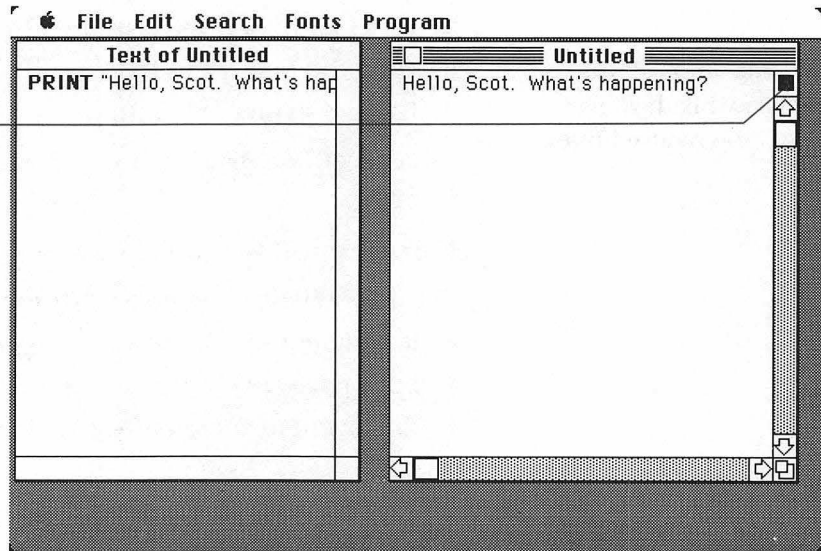
- *Assignment needs equals*—you spelled PRINT wrong.
- *Extra garbage*—you forgot the opening quote mark.
- *Ending quote not found*—you didn't type a closing quote.

**To Fix It**

For all cases:

1. Click the OK button in the error box:
  - a. Move the pointer to the button.
  - b. Press and release the mouse button.
2. Click the **close box** in the upper left corner (see Figure 2-1):
  - a. Move the pointer to the box.
  - b. Press and release the mouse button.
3. Press and hold down the **backspace** key until your error is erased.
4. Retype the line.
5. Choose Run from the Program menu.

Status Box—the large black square indicates program is finished



**Figure 2-2** Your first Macintosh BASIC program



To **click** in something means to move the pointer to some area and then press and release the mouse button.

Some error boxes have a cancel button at the bottom right. If you **click** (the Mac term for “press and release the mouse button”) on it instead of on the OK button, BASIC won’t check for typing errors until you run the program. I recommend that you don’t use this advanced feature until you have more BASIC programming experience.

### What “Run” Does

MacBASIC does a number of things when you tell it to **run** or carry out the instructions in a program. Much of its activity is concerned with finding mistakes. First it checks your typing for **syntax errors**; that is, it checks whether you’ve forgotten to include any keywords necessary to the operation of the program, and how your punctuation shapes up. If it finds any errors, it tells you right away; you’ll see how in a few minutes. Then it looks for **runtime errors**, mistakes it can’t find until it actually tries to carry out the instructions you’ve written. Assuming everything is all right, it produces an **output window** like the right-hand window in Figure 2-2, in which it displays any results the program produces. In this case, it produces the greeting

Hello, (whoever you are). What’s happening?



### For Your Information

**Don’t Memorize Anything** Don’t worry about the new vocabulary in the paragraph you just read. Memorizing definitions won’t help you learn BASIC. You’ll automatically pick up terms you need to know.



The only programmer who doesn't get coding errors is a programmer who doesn't type code into a computer.

### A Note on Errors

In the short space of this session I've talked about errors a lot. Programming and errors go together like politicians and campaign promises. Don't worry about errors or "bugs." You'll develop debugging skills as you develop other programming techniques. And, the more you program, the fewer mistakes you'll make. At the very least, your errors will become more sophisticated.

## Editing

---

The process of changing or adding to a program is called **editing**. You do it by means of a built-in program called the **Editor**. You'll probably use the Editor more than any other Macintosh BASIC tool. In the rest of this session you'll edit your program in a variety of ways, mostly using the mouse and the commands in the Edit menu. The Edit menu in MacBASIC holds the same commands that are in the Edit menus in all Macintosh applications; *Macintosh*, the owner's guide that came with your computer, gives all the details. This section describes the editing commands in the context of BASIC and gives you practice in using them.

In order to edit your program, you'll need to make the listing window active by clicking in it. Action can happen only in an active window; in MacBASIC, editing happens almost exclusively in the listing window.



### Do This

Make the listing window **active**.

1. Move the pointer until it lies anywhere within the listing window.
2. Click the mouse—that is, press and release the mouse button.

You know that the listing window is active because:

- the **title bar** has a close box and horizontal lines;
- the **vertical scroll bar** on the right side has up and down arrows;
- the **horizontal scroll bar** on the bottom has left and right arrows;
- a **size box** appears in the lower right corner;
- the pointer takes the shape of an **I-beam** when it's within the window.

There really isn't enough text in the window to do much editing on yet, so you'll need to add to the program. You could type in more PRINT lines, but you might as well let the computer do it for you by using commands from the Edit menu.

In the following exercise you'll make copies of the text that's already in the program by using the editing commands **Copy**, **Paste**, and **Select All**.



## Do This

---

Select the entire contents of the listing window and copy it to the **Clipboard**, a holding place for text and graphics that are being transferred from one place to another.

1. Choose Select All from the Edit menu (you'll see the whole line go from black-on-white to white-on-black).
2. Choose Copy from the Edit menu.
3. Move the pointer anywhere below the selected text in the listing window and click the mouse.
4. Choose Paste from the Edit menu.

Your screen should now look more or less like Figure 2-3.

## If Something Goes Wrong

- Text isn't Selected—you released the mouse button too soon.
- Text won't go into Clipboard—ditto.
- Text won't Paste—ditto.

## To Fix It

Repeat the step and hold the button down a bit longer.

"Program Finished" icon again.

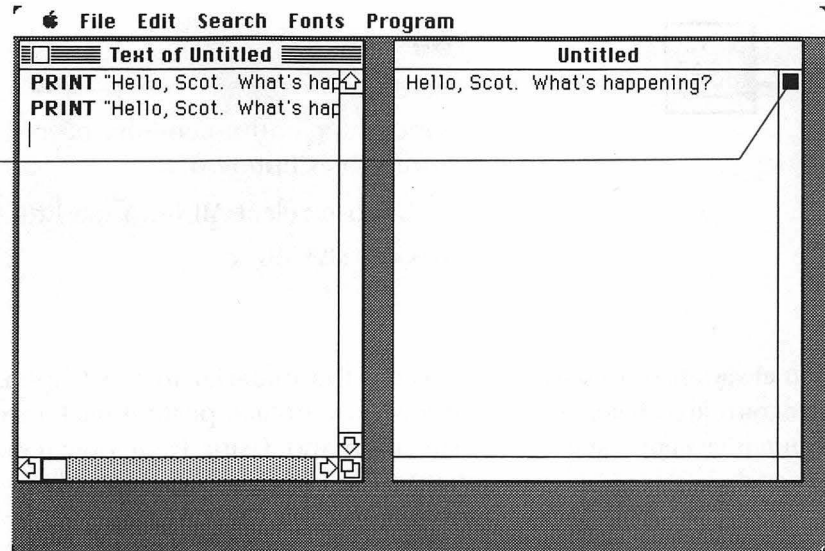


Figure 2-3 Result of Copy/Paste operation

## Select, Copy, and Paste

When you choose Select All, all the text in the listing window changes from black letters on a white field to white letters on a black field. This identifies it as **selected text**. Selected text is text ready to have something done to it by an editing command. When you choose Copy, a copy of the selected text goes into the Clipboard. (To see the Clipboard, you can choose Show Clipboard from the Edit menu—but it doesn't have to show in order to work.) Finally, when you choose Paste, a copy of the material in the Clipboard moves into the listing window at the **insertion point**, always marked by a blinking vertical bar, where typed or pasted material appears.

## Cut

When you choose Cut instead of Copy from the Edit menu, the selected material still goes into the Clipboard, but it also disappears from the listing window. Try it now, just for the experience.

The insertion point is the vertical bar marking the place where typed or pasted material appears.



### Do This

Remove the entire contents of the listing window and put it into the Clipboard.

1. Choose Select All from the Edit menu.
2. Choose Cut.

You always have to select program lines before you can cut or copy them.

Paste the material in the Clipboard back into the listing window now. In fact, paste it back three times by repeating the Paste command. Using Paste doesn't empty the Clipboard; material in the Clipboard remains there until you replace it with new material via Cut or Copy.



### For Your Information

**Active Clipboards**—You can make the Clipboard active, just as you can any other window. If you have trouble seeing what's in the Clipboard because other windows partly cover it, just click the mouse anywhere within it. Then you can move the pointer to its title bar and drag it around until you can read the Clipboard's contents clearly.

### Editing Letters, Words, and Phrases

The editing commands work only on selected text—text appearing as white characters on a black background—but you don't have to select all the text in the Listing window. The next exercise shows you how to select a specific part of the text using the mouse, and then change it.





## Do This

Change the word *happening* in the second line of the text to the phrase *going on*.

1. Select the word *happening* in the second line.
  - a. Move the pointer until it's over the *h*.
  - b. Press and hold down the mouse button.
  - c. Slide the pointer through the word until all the letters are selected.
2. Type the phrase *going on*.

Figures 2-4a and 2-4b give you before and after pictures for this process.

### If Something Goes Wrong

Too many or too few characters selected—you aren't used to rolling the mouse around yet (it takes time).

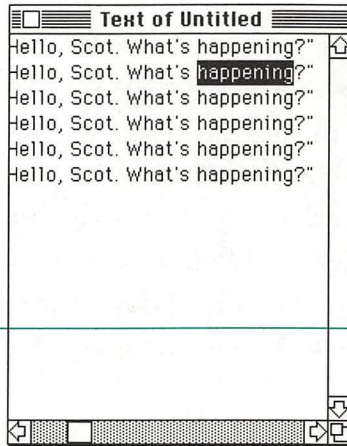
### To Fix It

Click the mouse button to deselect the text and start again.

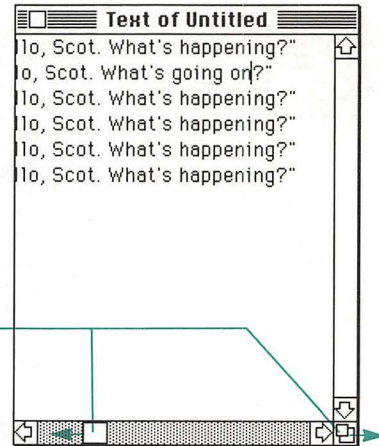
As soon as you begin to type over selected text, all the selected text disappears (it doesn't go into the Clipboard). What you type replaces it.

You can Paste as well as type to replace selected text. The next exercise gives you experience in using Cut on specific text, replacing the contents of the Clipboard, using Paste to replace selected text, and moving material from one part of a program to another.

To see all of the output drag the scroll box all the way to the left and drag the size box to the right.



**Figure 2-4a** Selected text before replacement



**Figure 2-4b** Result of replacing selected text



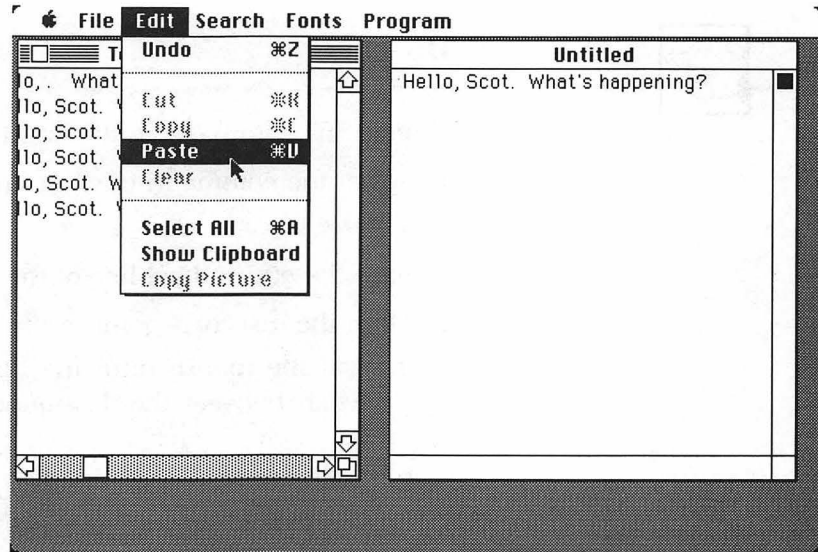
### Do This

Replace the word *happening* in the second line from the bottom with your name from the first line:

1. Select your name in the top line.
2. Choose Cut.
3. Select *happening* in the second line from the bottom.
4. Choose Paste.
5. Ponder the philosophical implications of the line into which you've just pasted your name.

Figure 2-5 shows step 4.





**Figure 2-5** A Cut/Paste operation in progress

### **Clear and the Backspace Key: Removing Without Replacing**

Sometimes you want to eliminate selected text from the listing window without either moving it to the Clipboard or replacing it with new material. You can do that either by choosing Clear from the Edit menu or by pressing the Backspace key. This next exercise gives you some quick practice in using both methods.



### Do This

Remove the comma from the first line.

1. Select the comma in the first line.
2. Choose Clear.

Remove the entire third line of the program.

1. Move the insertion point to the end of the third line.
  - a. Roll the mouse until the I-beam pointer is somewhere between the close quote and the vertical scroll bar.
  - b. Click the mouse.
2. Press and hold down Backspace until all the characters in the line disappear.

### If Something Goes Wrong

*Too many characters disappear*—you held down the Backspace key too long.

### Undo

Any material you take out of a program by using Clear or by pressing Backspace just disappears. It doesn't go into the Clipboard. Usually that's no great tragedy, but sometimes you don't want it to happen. The only way to get the material back is to use the **Undo** command immediately.

The Undo command lets you change your mind about some piece of editing you've just done. It replaces the often-heard "Oh my stars, why did I do that?" and similar less printable phrases and has been responsible for saving many computers that otherwise would have been hurled across rooms by frustrated programmers.

Undo undoes your most recent editing. It puts things back the way they were before your last command or series of keystrokes. This next exercise, the final one of this session, lets you wipe out your whole program and then get it back again.



If you want to undo an editing command, you must do it before you issue another command.

### Do This

Use the Backspace key to erase your program and Undo to get it back.

1. Choose Select All from the Edit menu.
2. Press Backspace.
3. Pretend the program you just wiped out was 1,200 lines long and was your only copy; feel panic.
4. Choose Undo from the Edit menu.
5. Experience relief as your program reappears.

### If Something Goes Wrong

*Program doesn't reappear*—you pressed the Backspace key another time or issued another command before Undo.

### To Fix It

You can't fix it. The program is lost. Go kick the dog.

## An Editing Shortcut

Before you go on to practice what you've learned, pull down the Edit menu. You'll notice that several of the commands are followed by the symbol ⌘ and a character. These symbol-letter combinations are called **Command key options**; you can use them to issue various commands from the keyboard rather than using the mouse. Here's how to use them:

1. Select the text or position the insertion point.
2. Press and hold down the ⌘ key.
3. Press a specific letter key (for example, X to Cut).



## Play Time

Spend some time experimenting with the editing commands you've learned in this session before you go on to the next one. Try different combinations of commands. Make lots of mistakes on purpose so that you can become adept at correcting them.

## Summary

---

### New Terms

---

**Active window** the window that your commands affect.

**Click** to press and release the mouse button.

**Close box** small box in upper left corner of active window. When clicked, it makes the window go away. Also called go-away box.

**Code** instructions written in a computer language.

**Command key option** option you use to give BASIC a command from the keyboard using the  $\mathbb{H}$  key rather than from a menu using the mouse.

**Debugging** process of locating and removing errors (bugs) from a program.

**Double-click** to press and release the mouse button twice.

**Drag** to move a window by positioning the pointer on its title bar and then holding down the mouse button while rolling the mouse around.

**Editor** Macintosh's built-in facilities for entering and modifying text and graphics.

**I-beam** shape of pointer when it appears in an area capable of being edited.

**Insertion point** vertical bar positioned where newly typed or pasted material will appear.

**Keyword** word or phrase having a specific meaning to the computer.

**Listing window** window holding listing of MacBASIC program.

**Menu bar** bar running across top of Macintosh display showing the names of and giving access to all available Macintosh commands.

**Output window** window displaying results of a MacBASIC program.

**Pointer** mouse's noseprint.

**Program** set of coded instructions that makes a computer do something.

**Quoted string** text enclosed in quote marks.

**Return key** key on right-hand side of keyboard; you press it to tell BASIC to act on what you've typed.

**Runtime error** error that BASIC can't find until you run a program.

**Select** to mark text for action by some command or keystroke.

**Shell** the Macintosh BASIC programming environment.

**Syntax error** error in keyword spelling or punctuation; any errors that occur when you type a program line. BASIC finds these errors when you press Return.

**Text** any collection of characters.

**Title bar** horizontal bar running across the top of a window showing the window's title and, for most active windows, a close box.

## Editing Commands

---

**Clear** remove selected material entirely.

**Copy** copy selected material to Clipboard.

**Cut** remove selected material; move to Clipboard.

**Paste** insert material from Clipboard into window at insertion point.

**Select All** prepare all material in window for some editing action.

**Undo** cancel effect of most recent keypress or command.

## File Commands

---

**Run** execute program in active listing window.

## Keys

---

**Backspace** delete selected material; delete character immediately before the insertion point.

**⌘** when used with other keys, allows you to give BASIC a command from the keyboard without using the mouse.

## Programming Statements

---

**PRINT** display some information in the output window.

## Bughouse

---

Both of the following program lines have errors in them. Figure out what's wrong with each line and fix the problem. Then type the fixed versions into your Macintosh to make sure they work correctly.

PRIN "I have always depended on the kindness of strangers."

PRINT "But yah are, Blanche, yah are."



## SESSION



### 3

# Introduction to Variables

---

**O**ne of the things that gives computers their great power is their ability to deal with **variables**, symbols that can stand for any one of a number of different values. In this session you'll learn how to use one type of variable, the **numeric variable**. You'll also learn the rudiments of computer arithmetic and how to show the results of computer arithmetic attractively on the screen. You'll have your first taste of interactive programming with the keyword `INPUT`, get more practice with `PRINT`, and learn how to add comments to your program. Finally, you'll store the program you've developed in this session on a disk.

Before you begin the session, make sure your BASIC desktop (the computer screen) is clear. If you have any experimental listing or output windows, click their close boxes. (A box may appear, asking if you want to "save changes"; just click "No.") Then get a new listing window by choosing New from the File menu.

## Computer Arithmetic

The simplest way to begin to describe variables is to look at computer arithmetic. Computer arithmetic looks very much like ordinary arithmetic with a few minor exceptions. Table 3-1 shows the four most common **arithmetic operators**, or symbols indicating arithmetic operations. The way they appear in everyday arithmetic is shown on the left and the way they appear in computer arithmetic is shown on the right.

**Table 3-1** Common Arithmetic Operators

	Everyday	Computer
add	+	+
subtract	-	-
multiply	×	*
divide	÷	/

Arithmetic formulas in Computerease look something like the constructs in algebra. Look at the following “problem” and try to figure out the answer:

If  $A = 5$ ,  $B = 10$ , and  $C = A + B$ , then what does  $C$  equal?

If you answered “That depends,” there’s probably a future for you in either politics or the law. On the other hand, if you answered 15, you’re on your way to understanding how to talk to your computer. Here’s how the problem looks in MacBASIC:

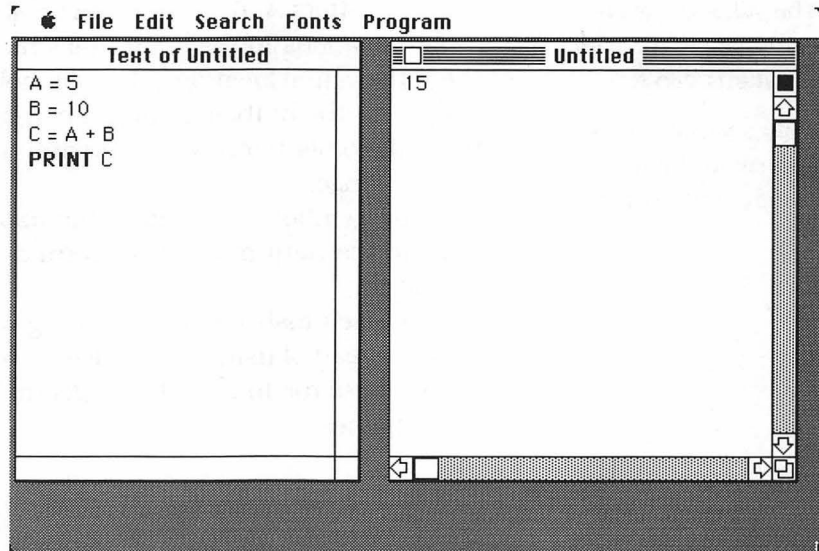


### Do This

Type the problem as a MacBASIC program:

```
A = 5
B = 10
C = A + B
```

That’s most of it. The only thing left is to get the machine to tell us what  $C$  is. As you learned in the last session, that’s what PRINT is for.



**Figure 3-1** Simple addition program



### Do This

Finish entering the program, run it, and compare the program to the results:

1. Add this line to the end of the program:

PRINT C

2. Choose Run from the File menu.

Figure 3-1 shows how things should look.

### Names of the Parts

Here are some definitions for the components of your program. Don't try to memorize them; they'll come up so often that you'll get to know them automatically.

The value of variables can change; the value of constants can't.

Only a variable can appear to the left of an assignment operator.

The letters A, B, and C are **variable names**. Variables represent locations in the computer's memory (you needn't worry about the actual locations) that can hold any value your program assigns to them; their values can change. The numbers 5 and 10, on the other hand, are called **constants** because their values never change.

The symbol = is called the **assignment operator**; any value to the right of it gets **assigned** (get it?) to the variable on the left.

In Macintosh BASIC, you can give variables any name you want. Instead of using letters like A and B (which remind mathophobes like me too much of algebra), you can use more meaningful names.



### Do This

Using the mouse and the keyboard, change A, B, and C to First.Value, Second.Value, and Sum.

1. Move the insertion point to the left of A.
2. Drag the mouse across A to select A.
3. Type the variable name First.Value.
4. Move the mouse to the left of B and change B to Second.Value.
5. Do the same thing for C and Sum.

Your program should end up looking much like Figure 3-2.



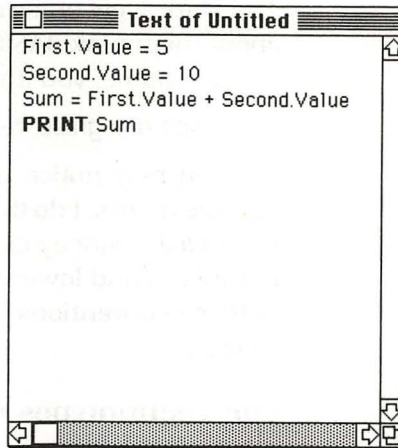


Figure 3-2 Substituting meaningful variable names

### Meaningful Variable Names

The third line is the heart of the program, where the computation happens. You can see that the new version of this line looks more like the way we humans think when we're solving problems than the old version did. In fact, the second version of the whole program is closer to the language we'd actually use and is far more meaningful than the first. Later, when you write complex programs, the usefulness of meaningful variable names will become immediately apparent to you.

Programs should read  
the way people think.

**Restrictions on Naming Variables** There are a few restrictions on the names you can give variables, however. Here's a list of them:

- The first character must be a letter of the alphabet.
- The name can't include colons, semicolons, commas, or spaces.
- The name can't include the symbols  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ ,  $>$ ,  $^$ , or  $=$  (collectively known as **operators**, about which more later).
- The name in many cases can't be exactly the same as a keyword, even if you use different capitalization. This means you can't call a variable PRINT or Print or PrInt, since BASIC doesn't differentiate between upper and lower case. (See Appendix B for a complete list of keywords.)



There are a few other, more obscure restrictions, but don't worry about them now. BASIC will tell you when you're using an illegal variable name; you'll get a message like

Can't recognize the rest of this line.

You may notice that I sometimes use a period within my variable names. I do this because it helps me read variable names more easily (since I can't use spaces). That's also why I use both uppercase and lowercase letters. You don't have to follow either of these conventions if you don't want to, just don't try to use spaces.

### For Technotypes Only: How Variables Work

BASIC maintains in its memory a **variable names table**. Table 3-2 gives you an idea of the way it works. When you assign a value to a variable for the first time, BASIC adds the name of the variable to the bottom of its list. Then it finds an unused location somewhere in the computer's memory and assigns that location to hold the current value of the variable (I say current because the value of the variable is likely to change). During the course of program execution, the program looks at that location whenever it needs to know the current value of that variable. If the value of that variable changes, the new value replaces the old one in the same location.

The location for the value of a given variable name remains constant: the value can change, but the location associated with the name won't. Every time the value for the variable changes, the new value is stored at the appropriate location and the old value is thrown away. The locations listed in Table 3-2 are just examples; the computer decides where to put the values, and there's no easy way to determine what the actual locations are.

**Table 3-2** Simulated Variable Names

Name	Location	Current Value
First.Value	2953	5
Second.Value	3657	10
Sum	3225	15



Programmers who “play computer” spend far less time debugging than those who don’t.

### Do This

Before going on, spend a few minutes changing the values of the variables `First.Value` and `Second.Value`.

First change just one variable. Then change just the other variable. Then change them both. Then change the arithmetic operator. Use all four of the ones you know. Each time you change a variable or an operator, predict what the result will be. Then have the computer run the program to check your prediction.

Each time you make a change, run the program *in your mind*, statement by statement, before you have the computer execute it. In other words, “play computer.”

“Playing computer” is a habit well worth developing. It will help you catch more errors than you can imagine.

## Making the Program More Flexible

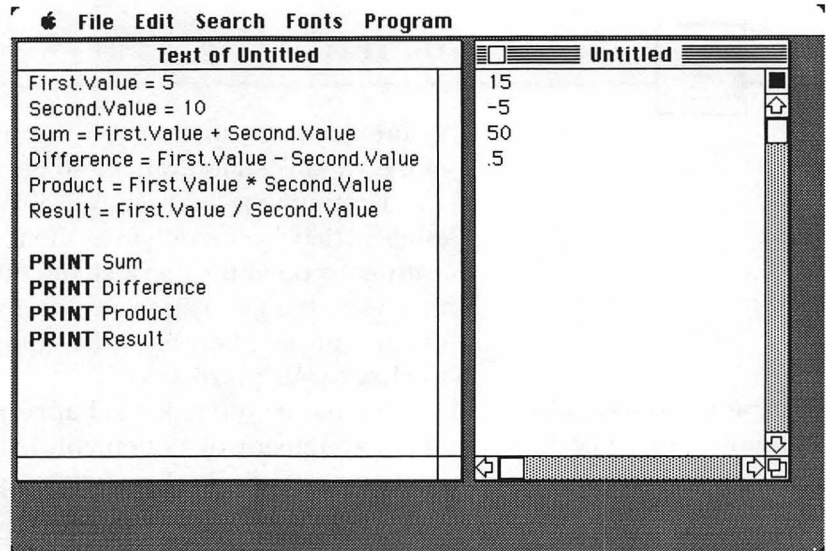
Figure 3-3 shows the program you’ve been working on, changed to work with all four arithmetic operations at the same time. I got the blank lines before the `PRINT` statements by pressing Return a few extra times.



### Do This

Make your desktop look like the one in Figure 3-3.

1. Type the new lines into your computer.
2. Widen the listing window by dragging the size box to the right.
3. Execute the program.
4. Use the size box to shrink and move the output window.
  - a. Grab the output window’s size box with the pointer.
  - b. Drag the mouse diagonally toward the upper left.
  - c. Grab the title bar and drag it diagonally toward the lower right.



**Figure 3-3** Four arithmetic operations

### Data vs. Information

Your program's variable table has just grown from three elements to six: you've added Difference, Product, and Result. Your program is now producing four pieces of information instead of one. But the output window just shows four numbers, without any indication of what they refer to. If you didn't have the listing window in front of you, you'd never know what the numbers mean—you'd have four pieces of data but essentially no information. It's like tuning in the 11 o'clock news and hearing "And now for the sports scores: 27 to 13, 16 to 5, and 26 all."

Information undescribed  
is no information at all.

In the next part of the session you'll learn how to use a variation of the PRINT statement to add descriptions to the information your computer produces. First, though, you need to learn about the semicolon.

### Punctuated PRINT

When you use it in combination with the semicolon (;), the PRINT statement can do more than you've seen so far. Here's how it works.

The most recent version of your program has four PRINT statements. Each time BASIC executes one of them, it displays the value of the variable it was told to print, moves the insertion point to the left edge of the window, and moves down one line. In computer terms, it issues a **carriage return** and a **line feed**. It's as if BASIC told the computer to press its own Return key. The semicolon tells BASIC not to issue a Return after it prints something.



### Do This

Add a semicolon to the end of each of the first three of your four PRINT statements and run the program.

1. Change the first three PRINT statements to look like this:

```
PRINT Sum;  
PRINT Difference;  
PRINT Product;
```

2. Run the program.

Your output window should display 15-550.5.

The semicolon lets you print several things on the same line, including combinations of variables, constants, and string literals. A **string literal** is anything enclosed in quotes. Your first program consisted of the keyword PRINT and a string literal:

```
PRINT "Hello, Scot. What's happening?"
```

To change the data your program produces into real information, you can use a series of variables, string literals, and semicolons in your PRINT lines. You can have more than one variable or string literal or semicolon on each line.





### Do This

Modify your program to make the output window more descriptive:

1. Change the PRINT statements to those shown in Figure 3-4.
2. Use the size box and title bar to move the listing window as necessary to see complete lines as you enter them (they might be too long for the window's current width).
3. Use the vertical and horizontal scroll bars as necessary to check your typing and to make it easier for you to "play computer."
4. Run the program.

Note the spaces between *of* and the close quote, before and after *and*, and before and after *is* in each of the PRINT statements. If you don't add these spaces, some of the words and numbers will squish together on the display when you run the program, making them hard to read. Try it both ways to see for yourself (it only takes a few seconds to change the code).

### Neatness counts!

Figure 3-4 shows what your program should look like when you're done and what executing the program produces. I'll leave it to you to go back and add periods to the end of each PRINTed sentence (hint: each period must be a string literal.)



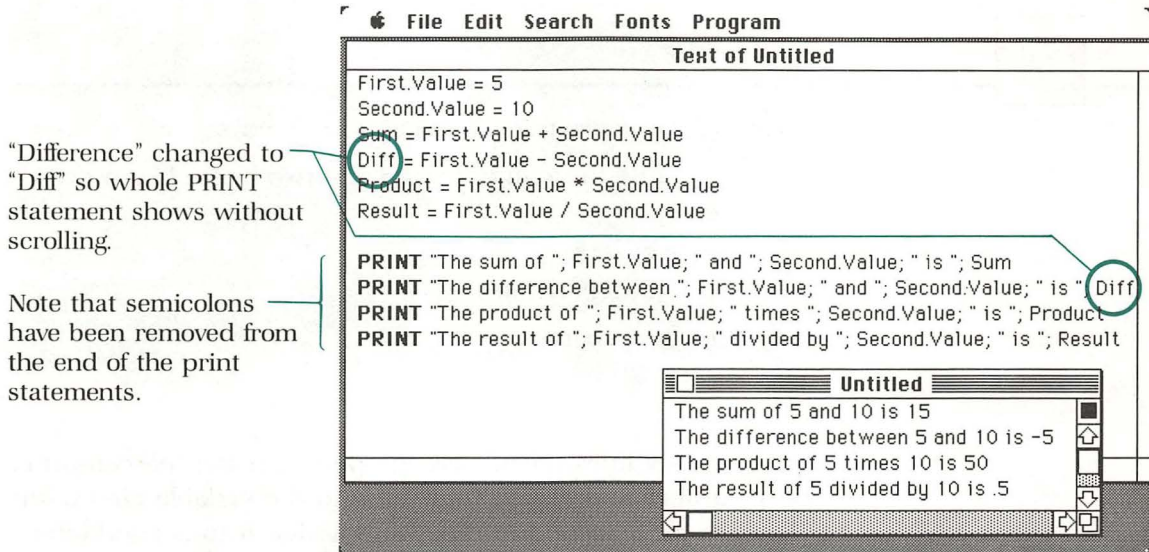


Figure 3-4 Descriptive PRINT statements

## INPUT: Getting Information from the Outside World

Earlier in this session you experimented with variables by changing the values for `First.Value` and `Second.Value`. The program would be a lot more useful if changing the values were easier—if there were some way you could give the program different numbers to operate on without having to mess with the mouse all the time.

Yes, you're being set up. The solution to this problem you didn't know you had is a new keyword, **INPUT**. **INPUT** lets the program accept information from the "outside world" while the program is running. The **INPUT** statement is the basis for **interactive programming**, programming that involves an ongoing exchange of information between the computer and the operator.



### Do This

Change your program to make it interactive.

1. Substitute these lines for the first two lines of your program.

```
INPUT First.Value  
INPUT Second.Value
```

2. Run the program.

Every **INPUT** needs an **INPUT** variable.

The new lines mean “Ask the person using the computer for a number and assign that value to the variable First.Value. Then ask for a second number and assign it to Second.Value.” In these statements, First.Value and Second.Value are **INPUT** variables.

When you run the program, a question mark and a blinking insertion point appear. The question mark is BASIC's way of letting you know it's waiting for you to type in something.

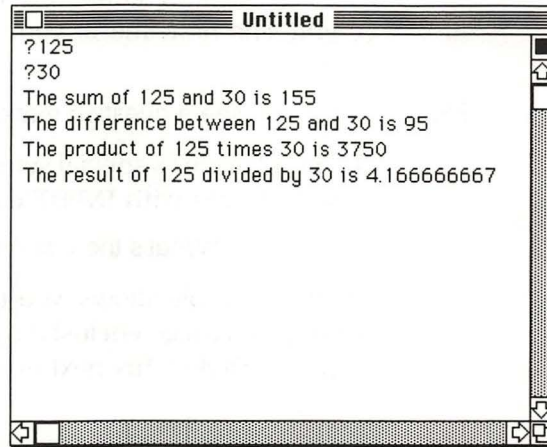


### Do This

Run the program again and respond to the computer's queries with the numbers 125 and 30.

1. Type the number 125.
2. Press Return.
3. Type the number 30.
4. Press Return.

If you're “playing computer” (which you should be), you might conclude that as soon as you press Return the first time, the variable First.Value gets the value 125—and you'd be right.



**Figure 3-5** Program modified to include INPUT lines

Pressing Return lets BASIC know that you're finished typing and that the program should continue. The INPUT statement for Second.Value generates the second question mark. When the program is finished, the output window should look like Figure 3-5.



### Do This

Run the program a few more times, using different numbers. Try minus numbers or decimals. Then come back to see how to make the program more fit for human consumption.

### INPUT Prompt Messages

Before you call in that Special Someone to see your first useful program, spend a few minutes **humanizing** it, or making it “friendlier” and easier to understand. You know that the question mark is BASIC’s way of telling you to type something—but your friend won’t. You need to come up with a more meaningful prompting message.

You already know one way to do that: Use a **PRINT** statement. The first line of your program might be something like this:

**PRINT** "When you see a question mark, type in a number and press Return."

**BASIC** provides a special syntax, however, to make using prompting messages with **INPUT** easy. It looks like this:

**INPUT** "What's the first number?"; First.Value

As the example shows, you type the keyword **INPUT**, an **INPUT** prompt message enclosed in quotes, a semicolon, and then the input variable. This next exercise lets you use both methods.



---

### Do This

Humanize your program with instructions and prompting messages.

1. Add a new first line that gives the operator instructions:

**PRINT** "Please type in a number and press the Return key."

2. Add prompting messages to the **INPUT** statements.

- a. Position the insertion point right after the **T** in **INPUT**.

- b. Add a space by pressing the space bar and type:

"What's the first number?";

- c. Position the insertion point right after the **T** in the second **INPUT**.

- d. Add a space and type:

"What's the second number?";

3. Run the program several times to make sure everything is OK.

Now you're ready to call your friend in. You'll still have to show your friend where the Return key is, of course, but doing that establishes you as the Computer Expert and is worth at least 25 prestige points.

Figure 3-6 shows my version of the completed program. I've added some extra spacing between lines and also a number of **comments**.

---

```
! Minicalc
! Version 1.0      6/5/84
! by John Scribblemonger
! Performs various calculations on values taken from operator.
!
!               Get the values
PRINT "Please type in a number and press the Return key."
INPUT "What's the first number?"; First.Value
INPUT "What's the second number"; Second.Value

!               Do the calculations
Sum = First.Value + Second.Value
Diff = First.Value - Second.Value
Product = First.Value * Second.Value
Result = First.Value / Second.Value

!               Display the results
PRINT "The sum of "; First.Value; " and "; Second.Value; " is "; Sum
PRINT "The difference between "; First.Value; " and "; Second.Value " is "; Diff
PRINT "The product of "; First.Value; " times "; Second.Value; " is "; Product
PRINT "The result of "; First.Value; " divided by "; Second.Value; " is "; Result
```

---

**Figure 3-6** Commented listing of Minicalc



---

## Program Comments (!)

---

The exclamation point (!) is a symbol BASIC understands to mean “everything from here to the end of the line is for humans.” BASIC ignores everything to the right of the ! on a line.

As you learn more BASIC keywords, your programs will get more complex. Sometimes even the best-written code can be hard to follow because of its complexity. However, **comments** allow you to see at a glance what a particular section of code is supposed to do. This is especially important when you haven’t looked at your program for a while.

Comments usually identify the program and its author, the overall purpose of each large section, and the specific functions of particular lines. Use extra spacing between sections of code to make program listings more readable and easy to follow.

A commented program is an understandable program six months down the line.



### Do This

Add your own comments and extra spacing to your program; more is better than less.

---

## Saving the Program

---

You’re finished with this program for now. It does something useful, so rather than just throwing it away you might want to store it on a disk until you need it again.



## Do This

Save the program to a disk under the name *Minicalc*.

1. Pull down the File menu.
2. Choose Save Text.
3. When BASIC asks for a name, type *Minicalc*.
4. Click the Save button.

## The Different Save Commands

There are two Save commands under the File menu, as you can see from Figure 3-7. The one you chose, Save Text, is the one to use the first time you save a new program to disk, while the program is still under development. The Save a Copy In... command lets you store the program you're working on under a name different from the one you originally gave it. That lets you keep several different versions of the same program on the same disk.

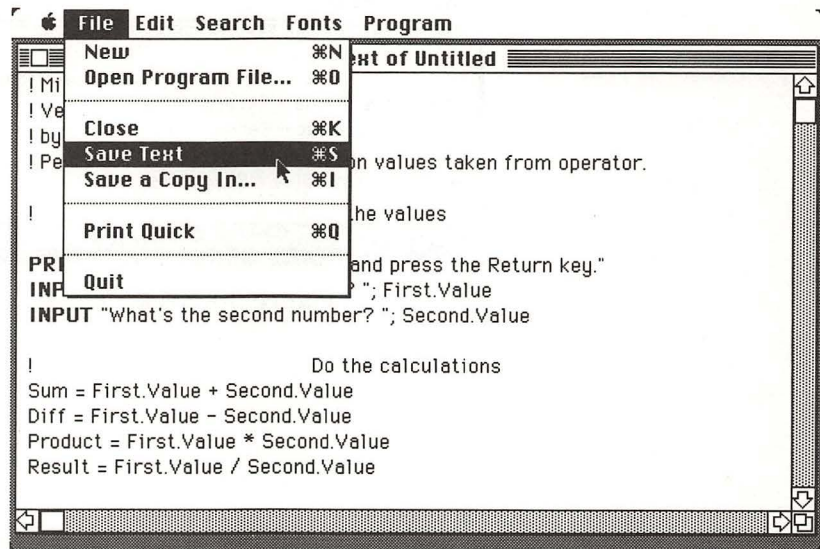


Figure 3-7 Save Text from File Menu

There's also a Save command under the Program menu. It's called Save Binary. This command stores the program in a form readable only by the machine; it strips away all the comments and extra spaces and converts your BASIC text to a much more compact form. Ordinarily you'd use this command only after you're totally finished with a program; most people never use it at all.

### The Dialog Box

The first time you choose Save Text to store an untitled program, a **dialog box** like the one shown in Figure 3-8 will appear. In the Macintosh, a dialog box shows up any time the machine needs some information from you.

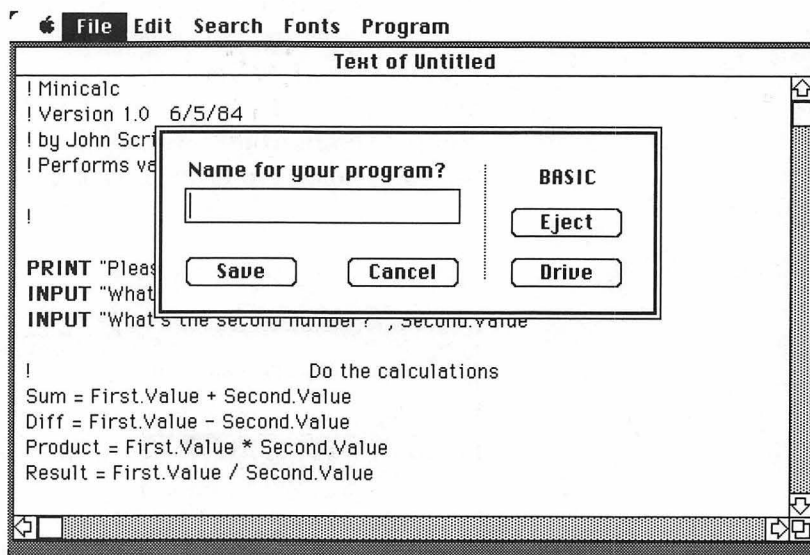


Figure 3-8 Dialog box for saving program

You have a number of options when this box appears. Cancel lets you go back to programming without saving the code. Eject kicks the disk out so that you can store the program on a disk other than the one in the drive. If you have a second disk drive attached to your Mac, Drive lets you choose where you'll store your program (if you don't have a second drive, the word Drive won't appear). **Save Text** stores the program on disk under the name in the listing window's title bar; if you haven't given the program a name yet, BASIC prompts you for one.

After you name and save your program, the dialog box goes away and the name of your program as listed in the title bar changes from Untitled to whatever name you chose. There now exists on the disk an exact copy of your program as it appears in the listing window (that's the entire program, even the part that you can't see unless you scroll it into view). It will stay on the disk in its current form, no matter what you do to the program in the listing window, until you save some other version of the program under the same name.

A program frequently saved is a program you don't have to type in—or create—again when your computer loses power.

It's a good idea to resave a program every so often when you're working on it. The Macintosh is a very reliable machine, but power failures and other catastrophes do happen. Experienced programmers resave their work every fifteen minutes or so during an active session.

Once your program has a name, the dialog box won't reappear when you choose Save Text; the program is automatically saved under the name appearing in the title bar.

### **All the Restrictions for Program Names**

You have to give your program a name if you want to store a copy of it onto a disk. The name can't have a colon (:) in it. That's the only rule. You can use any number, character, special symbol, spaces, or keyword that you want in a program name. Programmers tend to develop their own naming conventions and coding systems. My only recommendation is that you use names that will still mean something to you if you don't look at your disks for a year.





### For Your Information

**A Note on Upper and Lower Case:** Macintosh BASIC doesn't care whether you use uppercase or lowercase letters. It recognizes keywords, variables, and program names however you type them. The keyword PRINT is the same as PrInT or print, the variable Total is the same as total, and so on. The conventions used in this book reflect my own style. You can develop your own.

This is a good time to write a few interactive programs using INPUT (along with proper prompting messages) and to experiment with the different Save commands. Do it now—reinforce your learning.

## Summary

### New Terms

**Arithmetic operator** one of the four basic symbols for arithmetic operations (addition, subtraction, division, and multiplication).

**Assignment** the process of giving a value to a variable.

**Assignment operator** the operator =, used to give a value to a variable.

**Carriage return** movement of the insertion point to the left edge of the window; usually happens along with a line feed.

**Comment** a note to yourself (or to another programmer reading your code) that appears in a program line after the special symbol !. BASIC ignores comments; they're just for humans.

**Constant** that which never changes; generally applies to numbers to distinguish them from numeric variables.

**Desktop** another word for your Macintosh's screen, used because your Macintosh always makes available to you so many of the items you usually find on a desk top (an alarm clock, note pad, and so forth).

**Dialog box** a box that appears on the screen any time BASIC needs more information from you.

**Humanize** to make a program show its results or requests for information in polite and meaningful ways.

**INPUT prompt** optional string literal used in INPUT statements to let the operator know what information the computer needs.

**INPUT variable** variable at the end of an INPUT statement that accepts information from the computer operator.



**Interactive programming** programming characterized by an ongoing exchange of information between the computer and the operator.

**Line feed** movement of the insertion point down one text line; usually happens with a carriage return.

**String literal** anything appearing between quotes; same as quoted string.

**Variable** character or group of characters representing a location in the computer's memory where a value is stored. A variable can stand for any one of a number of different values.

## File Commands

**New** (⌘ N) create a new listing window on the desktop with the title Untitled.

**Save Text** (⌘ S) store a copy of the program in the active listing window to disk under a specified name.

## Programming Statements and Characters

+ addition operator

− subtraction operator

/ division operator

\* multiplication operator

= assignment operator

; used within a PRINT statement to prevent a carriage return/line feed.

! comment symbol; BASIC ignores all text between it and the end of the line.

**INPUT** get some information from the operator and store in the INPUT variable; display option INPUT prompt.

## Commands, Menu Items, Keywords You Know So Far

### Editing Commands and Keys

Backspace key

Clear

Copy

Cut

Paste

Select All

Undo

⌘ key

### File Commands

New

Run

Save Text

### Programming Statements and Characters

+ − / \* =

; ! INPUT PRINT

## Bughouse

---

There are five bugs in the following program that will generate error messages, along with a sixth one that makes a wrong answer appear (hint: the “wrong answer” bug is somewhere in the first three lines). Fix all the bugs you can find before you type the program into your computer; then type it in and run it to see if you really got all the bugs out.

```
First.Num = 5
12 = Second.Num
Sum = Firstnum & Second.Num
Print = Sum * 2
INPUT "What do you think Sum equals? " Sum.Guess
PRINT Sum.Guess " was your guess."
```

# SESSION

## 4

## Loops

**C**omputers are wonderfully obsessive-compulsive beasts. They can do the same task over and over again without getting tired or irritable. In fact, the ability to do rapid repetitions of the same task is one of the things that makes computers so useful. Showing you how to make your Mac repeat itself by performing what are called loops is one of the two major themes of this session. The other major theme is how to get the computer to *stop* repeating itself, using a technique called conditional branching.

### Some Loop Definitions

Every DO has a LOOP.

A **loop** is a series of program lines whose action repeats a number of times. That number can be either definite or indefinite, depending on the kind of loop you use. Loops by their nature change the **flow of control** in a program—the order in which BASIC executes program lines. If there are no loops in a program, control usually flows sequentially from the first line of code to the last; after BASIC carries out the last line of code, the program stops. The program you wrote in Session 3, *Minicalc*, is an example of a program with no loops.

## The DO\LOOP Construct

One of the simplest and most common loops in Macintosh BASIC is the DO\LOOP construct. The DO\LOOP starts with the keyword DO and ends with the keyword LOOP. All the statements between the keywords DO and LOOP repeat indefinitely.



### Do This

Type in a short program showing what DO\LOOP does.

1. Choose New from the File menu.
2. Enter this program:

```
DO
    PRINT "Danger—Runaway inflation!"
LOOP
```

3. Predict what running the program will do.
4. Choose Run from the Program menu.

Indenting makes reading code—and therefore debugging—easier.

You don't have to indent the PRINT line to make the program work, but it makes the code easier to read. You indent code either by pressing the Tab key or by pressing the space bar a few times.

If you've typed the code properly, your "inflation" line is running haywire like the one in Figure 4-1.

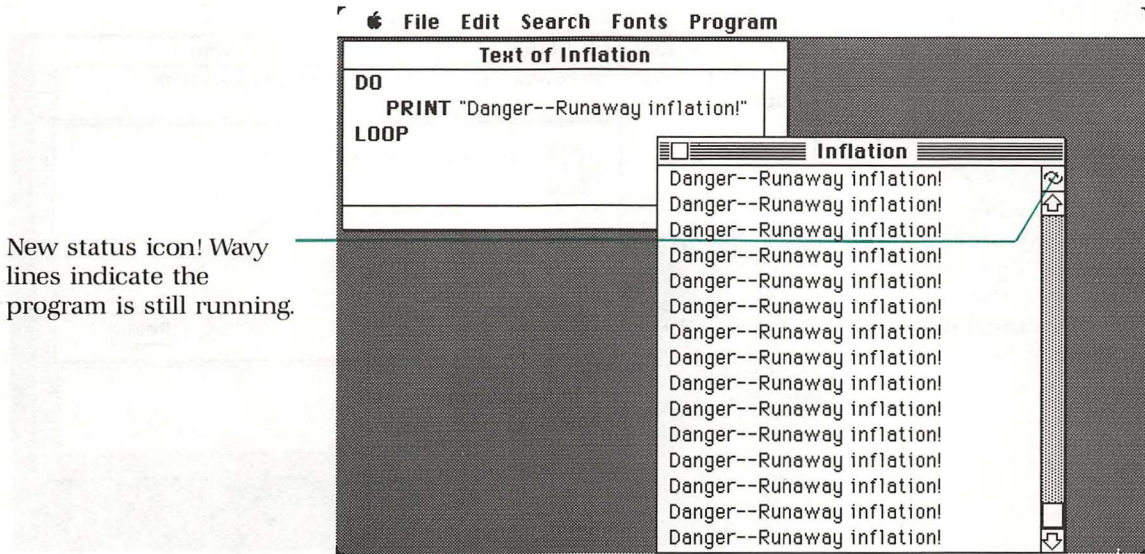


Figure 4-1 Inflation run amok



### For Your Information

**A Different Error Message** If you forget to add the key-word DO to your program, you'll get an error message like the one in Figure 4-2. It tells you about a Run-time error; one that BASIC can't detect until you run a program. If you click the button at the right marked Debug, BASIC turns on an advanced feature called the Debugger (you can also turn it on by choosing Debug from the Program menu). The Debugger helps you track down errors by, among other things, showing you in what order BASIC executes program lines. You don't need the debugger yet; but you'll want to experiment with it later when you write more complex programs.



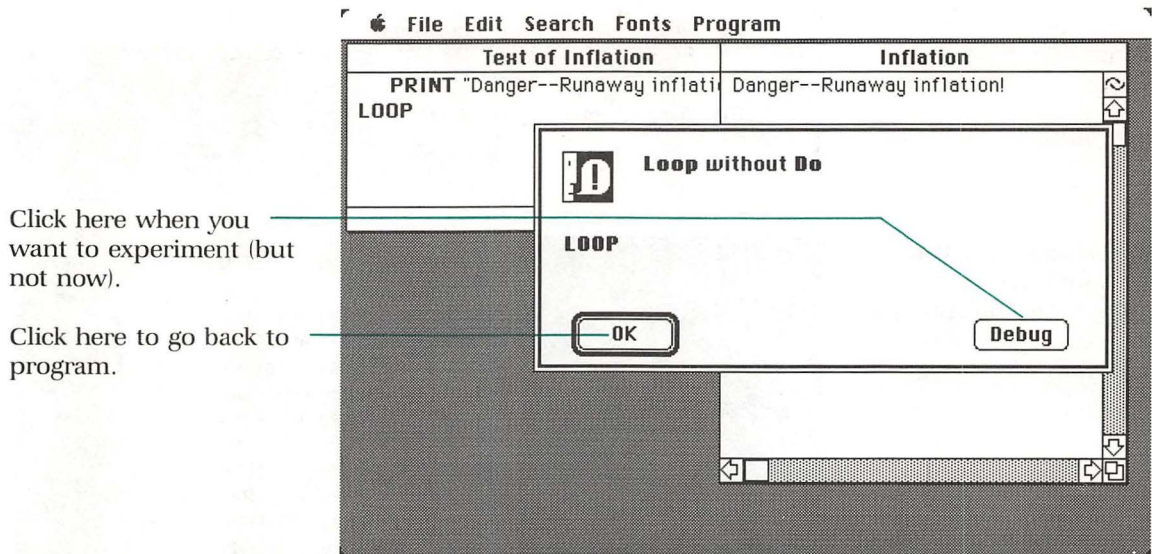


Figure 4-2 Error Message—"Loop without Do"



### Do This

Stop the program (thus saving the economy) by choosing Halt from the Program menu.

This little program demonstrates one of programdom's favorite bugs, the **infinite loop**. If it weren't for the Halt command (which stops any program in its tracks), this program would print the line "Danger—Runaway inflation!" right through to the next election.

This program, and nearly all programs using the DO\LOOP construct, would be much more useful if we could control the looping more. We need a way to make the looping stop after a certain number of **iterations** (number of times the statements within the loop's body are executed). That implies two things: first, that the computer can keep track of how many times it has done something; and second, that the program can escape the DO\LOOP. The next few paragraphs show you how to make your program do both.



### For Your Information

**About That DO\LOOP Backslash** The backslash means that some lines of BASIC code separate the keywords DO and LOOP. You'd never use DO and LOOP on the same line. You'll see this backslash convention several more times in this tutorial.

### A Loop Counter

In Session 3 you learned the basics of computer arithmetic and saw that computer arithmetic problems looked something like algebraic problems (as in  $C = A + B$ ). Algebra, however, would have a tough time with this line of code:

$$\text{Count} = \text{Count} + 1$$

The word *Count* is a numeric variable name; I might just as well have used any other name, but Count says what the line does. Each time BASIC executes this line, the value of Count increases by 1; in computer terms, the program **increments** the variable. You can think of the line as saying: "Let the variable Count hold the old value of Count plus one." This type of structure, in which some variable is incremented by some specific value each time BASIC executes the line, is (naturally enough) called a **counter**; you'll see it often.

The counter solves the first part of the problem—how to keep track of the number of times BASIC repeats the loop. If you add a counter to your program now and make a minor change to the PRINT statement, you can see the counter in action.



### Do This

Change the program so that it has a counter in the body of the DO\LOOP and so that it shows how many times BASIC has run the loop:

1. Move the insertion point to the left of the keyword PRINT.
2. Type the following line and press Return:

```
Count = Count + 1
```

3. Move the insertion point just to the left of the open quote in the PRINT line and type the word *Count* and a semicolon, like this:

```
PRINT Count; "Danger—Runaway inflation!"
```

4. Simulate the program in your mind—predict what the program will do when you run it. *Please don't skip this step!*
5. Run the program.
6. When the counter reaches 50 or so, stop the program by choosing Halt from the Program menu.

Unlooped lines don't repeat.

The counter works because it's inside the body of the loop. If you had placed the counter outside of the loop (before DO or after LOOP), the value for Count would always be 1. Try it right now and see.

### EXIT and IF...THEN: A Loop's Escape Hatch

It seems a shame continually to have to choose Halt when the program reaches a certain point. Luckily, BASIC offers the keyword EXIT and something called an IF...THEN statement to make the program handle the Halt chore for you.



The EXIT statement is very straightforward. When BASIC executes an EXIT statement, program control leaves the loop and resumes again at the first statement following the loop's bottom. In the DO\LOOP construct, since the bottom of the loop is marked by the keyword LOOP, BASIC jumps to whatever statement immediately follows LOOP. There isn't any statement following LOOP in the current version of your program, so, finding nothing else to do, BASIC just quietly ends the program.

Issuing the keyword EXIT by itself wouldn't be very helpful, however. Consider the following example, using the code you've already written.



### Do This

Simulate an execution of the following program in your head; you needn't bother typing it into the computer:

```
DO
    Count = Count + 1
    PRINT Count; "Danger—Runaway inflation!"
EXIT
LOOP
```

This program would increment the counter variable Count by 1, display the message

1Danger—Runaway inflation!

and then stop. As soon as BASIC executes the EXIT line, control branches out of the loop.

Not very interesting. But now change the EXIT line to read

```
IF Count = 25 THEN EXIT
```

and something much more useful happens.

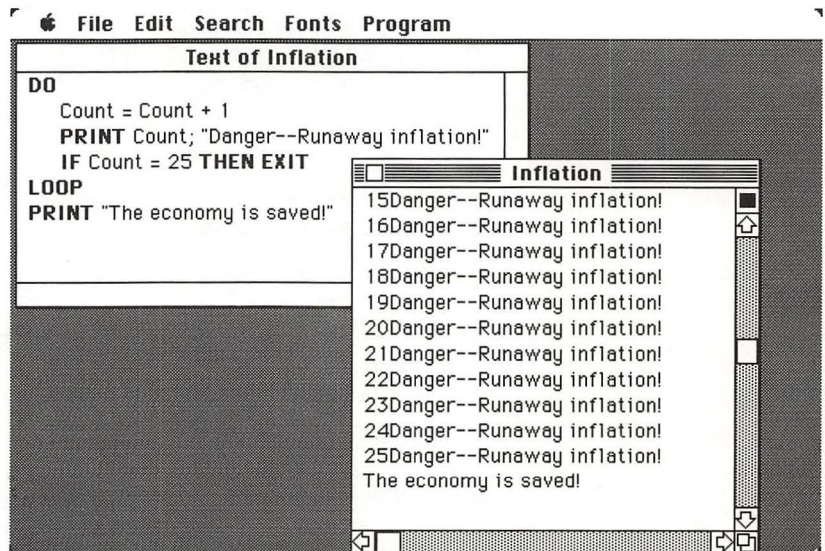


## Do This

Change the program in the output window so that it delivers its inflation message 25 times, displays some suitable concluding message, and then ends.

1. Insert the following line just before LOOP and press Return:  
IF Count = 25 THEN EXIT
2. Insert the following line after LOOP:  
PRINT "The economy is saved!"
3. Predict what will happen when you run the program.
4. Run the program.

Your program and its results should look like Figure 4-3.



**Figure 4-3** DO\LOOP with counter and IF...THEN EXIT



---

## IF...THEN: A Powerful Construct

---

IF...THEN is an extremely valuable construct in that it lets BASIC make decisions based on the outcome of a comparison. An IF...THEN statement says “IF some condition exists, THEN do something.”

In the program line you’ve just added, the condition that IF looks for is the outcome of the comparison between the value of the variable Count and the value 25; if the values are the same, BASIC **branches** to the statement following the keyword LOOP. This kind of branching is called **conditional branching**; it means that BASIC jumps to another part of the program when a certain condition is met.

### The Relational Operators

The IF...THEN EXIT line you just added makes BASIC leave the loop when Count holds 25. The comparison you make in that line between Count and 25 uses the symbol = to mean “the same value as.” There are several other symbols you can use to make comparisons; collectively these symbols are called **relational operators**.



### For Your Information

**This “=” Isn’t Assignment** You’ve used the symbol = to assign a value to a variable. This new use of = doesn’t assign, however—it compares. As a comparison maker, = can have either a variable or an expression to both its left and its right; as an assigner of values, it *must* have a variable to its left. These examples show = used in comparisons, as a relational operator.

```
J = 5  
IF J = 5 THEN PRINT “Yes”
```

Let your context be your guide. If the = is in a statement indicating a decision or condition, such as an IF...THEN statement, it is being used to compare.

Relational operators always compare two values.

Table 4-1 shows a list of the relational operators Macintosh BASIC uses. As you work through this tutorial, you'll find more and more situations where comparing two values comes in handy. As luck would have it, such a situation appears in the very next section.

**Table 4-1** Relational Operators

Symbol	Meaning	Example
=	equal to	Truth = Beauty
>	greater than	More > Less
<	less than	Less < More
<> or ><	greater or less than	This <> That
>= or =>	greater than or equal to	Officer >= Ensign
<= or =<	less than or equal to	Salary <= Expenses

Assume that the examples listed on the right have been assigned values that accurately reflect their names—for example, that variable Less holds a value less than the variable More.

### IF...THEN with Other Keywords

EXIT isn't the only thing that can follow the keyword THEN. Any valid Macintosh BASIC statement can do so. For example,

```
IF Sum > 500 THEN PRINT "Too much"
```

means "If the value held by the variable Sum is greater than 500, then display the words *Too much*." Similarly, the statement

```
IF Highest < Newguess THEN Highest = Newguess
```

means "If the variable Highest holds a value less than that of the variable Newguess, then assign the value of the variable Newguess to the variable Highest."

You'll come back to the *Inflation* program in a few minutes; just leave it where it is for now (if you're nervous about losing it, feel free to store it on disk under the name "Inflation"). Meanwhile, here's an example of IF...THEN used outside of a DO\LOOP.



### Do This

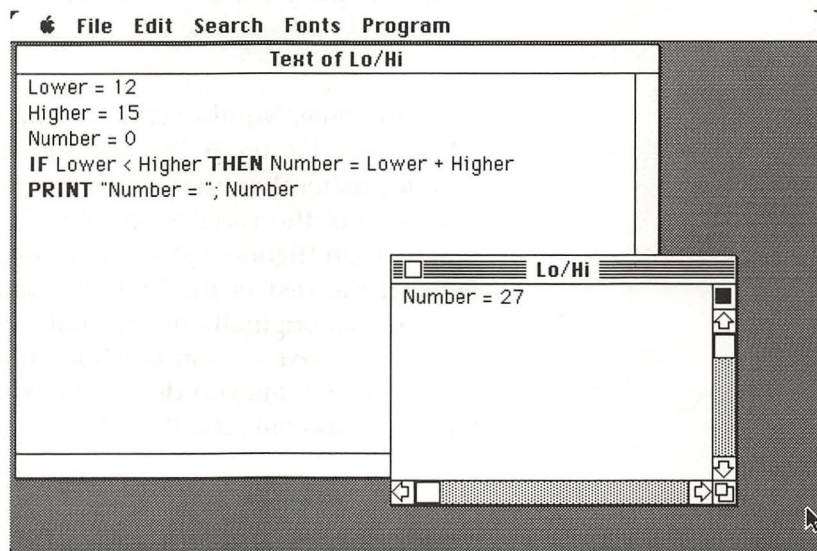
Enter this program into your computer; predict the results, and then run the program.

1. Choose New from the File menu.
2. Enter, but *don't* execute, the following program:

```
Lower = 12  
Higher = 15  
Number = 0  
IF Lower < Higher THEN Number = Lower + Higher  
PRINT "Number = "; Number
```

3. Predict what will appear in the Output window.
4. Run the program to prove that you're right.

The results should look like Figure 4-4.



**Figure 4-4** Program with relational operator



The variable `Lower` holds a value (12) that is less than the variable `Higher` (15). When BASIC comes to the IF line, it compares these two variables and sees that `Lower` is indeed less than (`<`) `Higher`. Since that's true, it goes on to execute the part of the line that comes after the keyword `THEN`: it resets the value of `Number` to be the sum of `Lower` plus `Higher`. Finally, it displays the new value for `Number`—27.

Now change the program slightly to see a different result.



### Do This

Predict what value `Number` would hold if the symbol `<` were changed to `>`:

1. Make the listing window active.
2. Change the “less than” symbol (`<`) to the “greater than” symbol (`>`).
3. Predict what will happen when you run the program.
4. Run the program to see if you are right.

This time, `Number` ends up holding its original value of 0. BASIC looked at the IF line and saw “If the variable `Lower` holds a value greater than the value of the variable `Higher`, then change the value of the variable `Number`.” But `Lower`'s value was *not* greater than `Higher`'s value—12 is not greater than 15—so BASIC ignored the rest of the line. The value for `Number` remained what it was originally, 0, and that's what got displayed.

In the next section you'll learn about another variation of `IF...THEN` that lets you determine what BASIC should do when a comparison fails the IF test.

### IF...THEN...ELSE

In the last IF...THEN example, when BASIC determined that a condition was false—that is, when a comparison failed the IF test—BASIC ignored the rest of the current line and went on to execute the next line. By adding the keyword ELSE and another statement to the end of the IF line, however, you can tell BASIC to do something else before it skips down.



#### Do This

Change the IF line to add an ELSE clause:

1. Make the listing window active.
2. Move the insertion point to the end of the IF...THEN line.
3. Add to the line:

```
ELSE PRINT "No change"
```

4. Predict what will happen when you run the program.
5. Run it.

When you add an ELSE clause to an IF...THEN line, the line means "IF the condition is true, THEN do something; if it isn't true, do something ELSE before going on." In the preceding example, the condition wasn't true (12 still isn't greater than 15), so BASIC ignored the THEN part of the line and went right to the ELSE part. It displayed the message "No change" (as instructed by the ELSE) before going on to show the value of Number. If the condition had been true, BASIC would have executed the THEN part and ignored the ELSE.

I'll leave it to you to change the > back to <, just to prove that what we both know will happen actually does.



## About the Scroll Bar

The output window of the *Inflation* program (which, I presume, is still someplace on the screen—no doubt buried behind a bunch of windows) has enough material in it now to present you with a feature of the Macintosh you might not have experienced yet: scrolling through material using the **scroll bars**. The scroll bars let you see material in a program listing or, as in this case, material a program has produced but that has scrolled out of sight. In Macintosh BASIC, when a window has more material than can fit in the viewing area, a **scroll box** appears in the scroll bar. The scroll box shows at the bottom of the bar in Figure 4-1, for example, indicating that there's "hidden material." By using the mouse, you can see what's scrolled out of sight.



### Do This

Scroll through the output window of the *Inflation* program using the mouse and the scroll arrows.

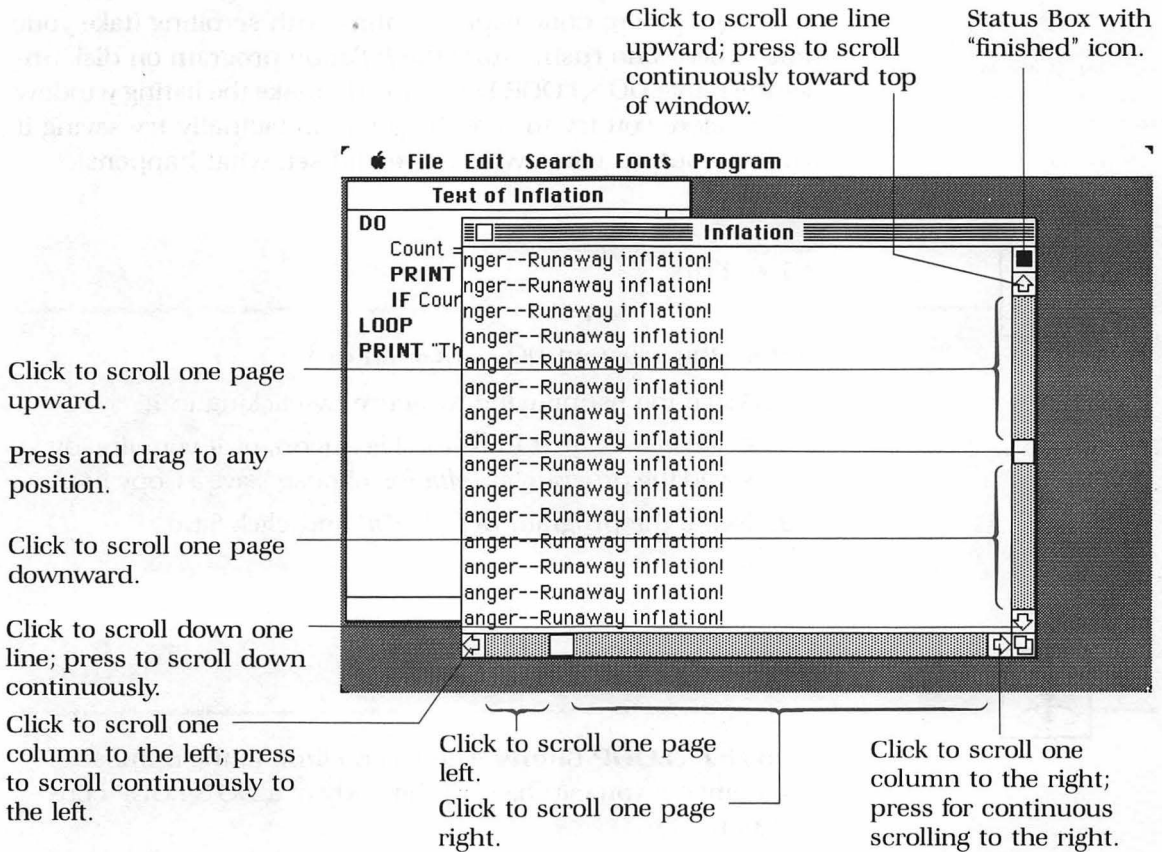
1. Make the output window of the *Inflation* program active.
  - a. Drag any windows in front of the *Inflation* output window out of the way.
  - b. Click anywhere in *Inflation*'s output window to make it active.
2. Position the pointer on the up-arrow at the top of the scroll bar.
3. Press and hold down the mouse button.
4. Position the pointer on the down-arrow.
5. Press and hold down the mouse button.

Actually, there are a number of ways you can scroll through almost all Macintosh windows—continuously as you've just done, a line at a time, or a viewing area at a time. You can also scroll directly to a particular section of the material.

Figure 4-5 shows a number of the Macintosh's scrolling features.

To scroll one line at a time, position the pointer on top of the appropriate arrow, depending on the direction in which you want to scroll, and click the mouse button.

To scroll a whole viewing area's worth, click in the gray area of the scroll bar either above or below the current position of the scroll box, again depending on the direction in which you want to scroll.



**Figure 4-5** Ways to scroll contents of a window

For ultimate scrolling control, position the pointer within the scroll box and drag it up and down; when you release the mouse button, the text in the output window will readjust. If the box is half-way up, for example, the center part of the output material appears; if it's at the top, the start of the output material appears.

The horizontal scroll bar across the bottom of the window works the same way; it's for material that's written beyond the right edge of the window. You've already seen that when you type past the right edge of the window, BASIC scrolls the text so you see what you're typing as you go along.

When you're done experimenting with scrolling (take your time—there's no rush!), store the *Inflation* program on disk under the name `DO\LOOP`. Remember to make the listing window active before you try to save the program (actually, try saving it while an output window is active and see what happens).



### Do This

Store the program `DO\LOOP` on disk.

1. Make the listing window active by clicking in it.
2. Choose Save Text from the File menu, or if you already saved the program as *Inflation* choose "Save a Copy In...".
3. Name the program `DO\LOOP` and click Save.



### For Your Information

**No `DO\LOOP` Limits** There is no limit to the number of statements you can have in the body of a `DO\LOOP` construct.





### Pop Quiz

Here's a quick quiz you can use to test how well you've learned the material in this session. Change the *Minicalc* program so that

1. It tells the operator what it does.
2. It asks the operator how many sets of numbers he or she wants to calculate.
3. It performs the calculations for as many sets as the operator indicated.
4. It delivers a final "goodbye" message of some sort.

In order to modify *Minicalc*, you'll have to learn a new command—Open.

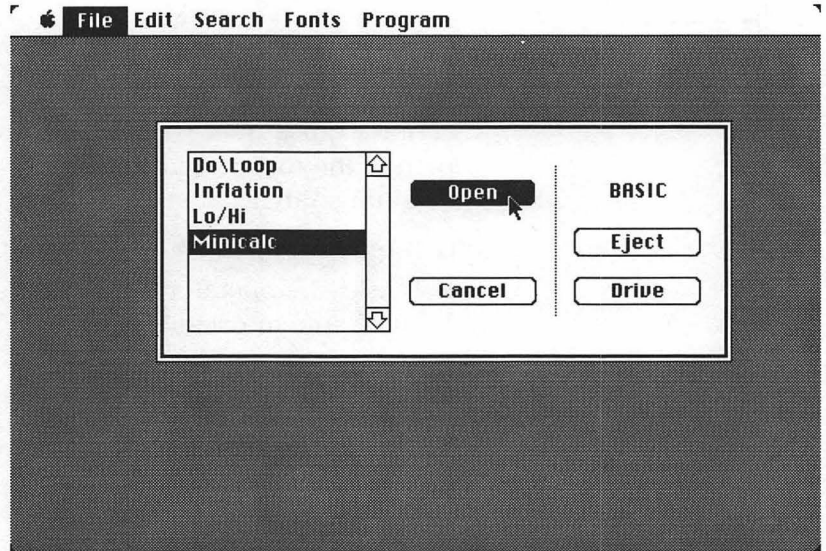


### Do This

Call back the *Minicalc* program so that you can change it to make it more flexible.

1. Choose "Open Program file" from the File menu.
2. Click *Minicalc* from the **Directory** that appears (see Figure 4-6).
3. Click Open.





**Figure 4-6** Opening the Minicalc program from the File Menu

Since this is your first quiz, I'll give you some hints. You'll probably need to use a counter and all of the following statements: PRINT, INPUT, IF...THEN EXIT, DO\LOOP.

The way I modified *Minicalc* is listed at the end of the session under (what else?) "Pop Quiz Answer."

## Summary

---

### New Terms

---

**Conditional branch** transfer of flow of control from one part of the program to another based on the outcome of some comparison.

**Counter** a variable used to keep track of loop iterations.

**Directory** list of all BASIC programs on the disk. The Directory appears when you use the Open Program file command.

**Expression** any group of numbers and/or variables coupled with operators and meant to be taken as a single entity; also, everything that appears to the right of the assignment operator =.

**Flow of control** order in which BASIC executes program lines.

**Increment** to increase the value of a variable by a specific value. You usually increment a variable repeatedly in a loop.

**Infinite loop** endless repetition of one or more program lines.

**Iteration** one pass through a loop, during which BASIC executes all statements within the loop.

**Loop** one or more program lines whose action BASIC repeats a number of times.

**Relational operator** one of the three basic symbols =, <, >, used either singularly or in combination to test the relationship between two values.

**Scroll bar** vertical or horizontal bar that you use with the mouse to see hidden text.

**Scroll box** small hollow box that appears in a scroll bar to let you know that some text is hidden.

### Menu Commands

---

**Halt** (⌘ H) stop the program whose output window is active (under Program menu).

**Open Program file...** (⌘ O) bring into memory the program whose name you choose from a list that BASIC shows you (under File menu).

### Programming Statements and Characters

---

= relational operator meaning "has the same value as."

> relational operator meaning "greater than."

< relational operator meaning "less than."

**DO\LOOP** repeat indefinitely, in order, the program lines falling between the keywords DO and LOOP.

**ELSE** when used at end of an IF...THEN line means execute the statement between this keyword and the end of the line if the preceding condition evaluates as false.

**EXIT** transfer program control out of the loop structure to the first statement following the bottom of the loop.

**IF...THEN** execute the statement between the keyword THEN and the end of the program line (or, if present, the keyword ELSE) if the condition between the keywords IF and THEN is true.

## Pop Quiz Answer

---

Here's the way I modified *Minicalc*. I did it in two steps:

1. Add just before the "Get the Values" section:

```
PRINT "This program adds, subtracts, multiplies, and divides pairs of numbers."
INPUT "How many pairs do you want to calculate? "; Pairs
PRINT
DO
```

I use PRINT to generate a blank line on the display, just to be neat (have I said that neatness counts?).

2. Add to the end of the program:

```
Loopcount = Loopcount + 1
IF Loopcount = Pairs THEN EXIT
PRINT
LOOP
PRINT
PRINT "Thanks for operating me!"
```

The really key line is the fifth from the end.

```
IF Loopcount = Pairs THEN EXIT
```

Here BASIC compares the value of one variable to the value of a second. If both values are the same, BASIC leaves the loop; if the values are not the same, BASIC executes the loop again.

If you did something else and it works, then your method is at least as good as mine.

## Commands, Menu Items, Keywords You Know So Far

---

### Menu Commands/Editing Keys

Backspace key	Paste
Clear	Run
Copy	Save
Cut	Select All
Halt	Tab key
New	Undo
Open	⌘ key

### Programming Symbols

+	*	!
-	=	>
/	;	<

### Programming Keywords

DO\LOOP	IF...THEN
ELSE	INPUT
EXIT	PRINT

## Bughouse

---

The following scrap of code is supposed to print the message "What's Happening?" 10 times. It has more bugs than a South Bronx tenement. Make it work right.

```
Printout = Printout + 1
DO THIS
  PRINT "What's Happening?"
  IF Print.Out = 10 LEAVE
REPEAT LOOP
```

# SESSION



## 5



# Controlled Loops

---

**I**n the last session you had your first look at a programming **control structure**, the `DO\LOOP`. The `DO\LOOP` is a control structure because it has a definite beginning (`DO`), a definite ending (`LOOP`), and consistent activity within its body (all statements repeat). In this session you'll work with another looping control structure, the `FOR\NEXT` loop.

While you're here, you'll also learn about string variables, which let you use symbolic names to stand for groups of text characters (such as letters and special characters); you'll add the keyword `CLEARWINDOW` to your vocabulary (it lets you clean up messy output windows); and you'll see what it's like to plan a program.

Spend lots of time on this session; don't rush through it. You'll use controlled loops constantly throughout your programming career. Experiment more than ever!



## FOR\NEXT: More Control, Greater Complexity

The **FOR\NEXT loop** and the **DO\LOOP** construct are alike in that they're both looping control structures, but **FOR\NEXT** gives you more control over the number of times BASIC executes the loop. **FOR\NEXT** has its counter built into its starting line. Here's what the syntax of the loop looks like:

```
FOR loop.variable = starting.value TO ending.value STEP step.value  
    statement(s)  
NEXT loop.variable
```



### For Your Information

**Syntactic Outline** The preceding weirdness is a **syntactic outline** of the **DO\LOOP** construct. A syntactic outline shows the form of a construct; the words in uppercase letters are keywords, those in lowercase are descriptive phrases. Many computer programming manuals (including the Macintosh's) use syntactic outlines of one kind or another to make descriptions clearer. The idea is that, once you understand the components of a syntactic outline, learning a new statement or construct is easy: you just look at the outline and immediately (supposedly) understand how to use it. I occasionally use syntactic outlines in this tutorial so that you'll recognize them when you come across them in other books and manuals.

To see what this syntactic outline is about, you'll need to type in a short program.



### Do This

Enter a short program to see how a FOR\NEXT construct works.

1. Clear out any leftover windows from previous sessions.
2. Get an Untitled listing window by entering ⌘ N.
  - a. Press and hold down the ⌘ key.
  - b. Press the “n” key.

(This is the same as choosing New from the Program menu.)

3. Type in this program:

```
PRINT "Before the loop"
FOR Sample = 1 TO 10 STEP 1
    PRINT "This is iteration #"; Sample
NEXT Sample
PRINT "After the loop"
```

4. You don't have enough information to predict with any certainty what will happen when you run this program, but take a shot at it anyway.
5. Run the program.

Figure 5-1 shows the results you should get. Compare what the program produces with both the program listing and the syntactic outline before reading any further; try to figure out what happened.

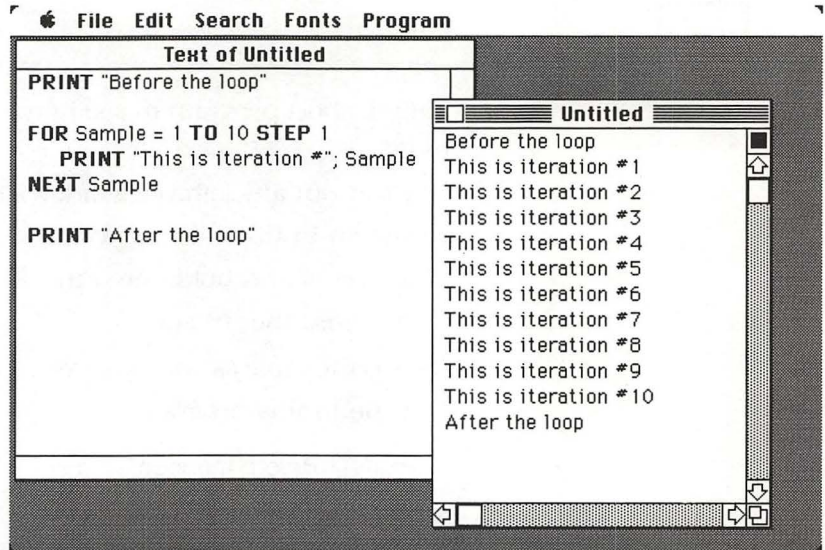


Figure 5-1 FOR\NEXT loop

### How It Works

The FOR line says "Establish a range of values FOR a variable called Sample, going from a starting value of 1 TO an ending value of 10, changing in STEPs of 1. If the value for Sample doesn't exceed the ending value (in this case, 10), execute the statements between here and the NEXT line." Note that Sample is a **loop variable**—a variable whose value changes on each pass through the loop. (Some people call this an **index variable**.)

The NEXT line says "Get the NEXT value for the loop variable Sample by increasing Sample's value by the value of STEP (in this case, 1—it works like  $\text{Sample} = \text{Sample} + 1$ ). Go back to the top of the loop to check whether BASIC should execute the loop again."

Every FOR starts a loop;  
every NEXT ends it.

So BASIC executes the loop 10 times, with the value of Sample changing from 1 through 10, incrementing by 1 on each **pass** (iteration). BASIC showed the phrases "Before the loop" and "After the loop" only once each because their PRINT lines occurred outside the loop.





### For Your Information

**Loop.Variable = Loop.Variable + 1** The final value for Sample is actually 11, because the NEXT line increments the loop variable until it exceeds the ending value. If you like, you can prove this to yourself by running the program again with an added final line:

```
PRINT "The final value for Sample is "; Sample
```



### For Your Information

**For Those Who Compare . . .** If the program we've been discussing ("we" being you, me, and the mouse) were written using a DO\LOOP instead of a FOR\NEXT construct, it would look like this:

```
PRINT "Before the loop"
Sample = 1
DO
    IF Sample > 10 THEN EXIT
    PRINT "This is iteration #"; Sample
    Sample = Sample + 1
LOOP
PRINT "After the loop"
```

### An Implied STEP

Actually, you could have left out the STEP clause in the FOR line; if you don't include STEP, BASIC assumes you want a STEP value of 1. Here's a sample program that demonstrates this shortcut and may also help clear up any lingering questions about how the value of the loop variable changes.





### Do This

Enter another program to demonstrate a FOR\NEXT loop with a different range of values.

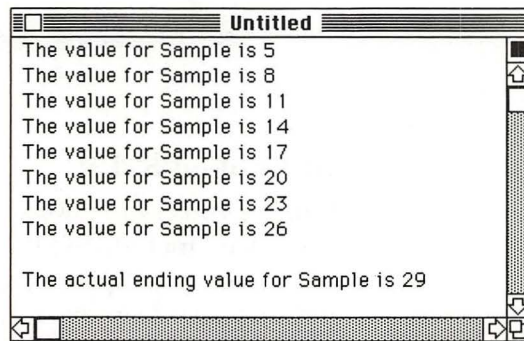
1. Get a clean listing window.
2. Enter the following program:

```
FOR Sample = 6 TO 10  
    PRINT "The value for Sample is "; Sample  
NEXT Sample
```

3. Predict the results.
4. Run the program.

### Using Variables in the FOR Line

You can express the starting, ending, and step values in the FOR line as variables (as you might have suspected). You can get the values for the variables from INPUT statements. The final surprise: you can use minus numbers! Use the following sample program to test my outrageous claims (and to discover other goodies):



**Figure 5-2** FOR\NEXT loop using a STEP value of 3



## Do This

Change the program in the last example so that it uses variables instead of constants in the looping structure.

1. Use the various commands from the Edit menu to change your program to look like this:

```
Start = 5
Finish = 28
Increment = 3

FOR Sample = Start TO Finish STEP Increment
    PRINT "The value for Sample is "; Sample
NEXT Sample

PRINT
PRINT "The actual ending value for Sample is "; Sample
```

2. Predict the results and run the program. Your output should look like Figure 5-2.
3. Convert the three variable assignment lines to INPUT statements and add an empty line for neatness, like this:

```
INPUT "What's the opening value? "; Start
INPUT "What's the limit? "; Finish
INPUT "And what's the increment? "; Increment
PRINT
```

4. Run the program, supplying whatever numbers you want.
5. Run it again, but this time answer the prompts with these values:
  - a. For the opening value: 20
  - b. For the limit: 5
  - c. For the increment: -3 (that's right: -3)

6. Run it yet again, using these negative numbers:
  - a. For the opening value:  $-10$
  - b. For the limit:  $-20$
  - c. For the increment:  $-5$
7. Run it one final time, using these decimal numbers:
  - a. For the opening value:  $.7$
  - b. For the limit:  $5.2$
  - c. For the increment:  $.8$



### Pop Quiz

Here are a couple of questions to see how well I've been doing my job. They call for a clear understanding of the way FOR\NEXT works. Half credit for you just for understanding the questions on the first reading; no credit for me if you don't understand them by the third.

#### Question 1

If you give an increment value of 0, your program will loop forever, printing out over and over "The value for Sample is " whatever you gave as the opening value. How come?

#### Question 2

If the first line of the loop says

FOR Iteration = 1 TO 0

BASIC won't execute the loop at all. Why?

You'll find the answers at the end of the session.

### EXIT in a FOR\NEXT Loop

You can get out of a FOR\NEXT loop early—before the loop variable's value exceeds the limit value—the same way you escape from a DO\LOOP: by using EXIT. The program below shows you how to do it.



#### Do This

Type in the sample program demonstrating how to use EXIT in a FOR\NEXT construct.

1. Set up a new listing window.
2. Enter this program:

```
FOR Iteration = 1 TO 100
  Sum = Sum + Iteration
  IF Sum => 15 THEN EXIT
NEXT Iteration
PRINT "Left loop on pass #"; Iteration
```

3. Predict what value the variable Iteration will hold when BASIC executes EXIT.
4. Run the program.

### Counter as Totaler

The above example has two variables that change on each pass through the loop. One variable, Iteration, is the loop variable. The other variable, Sum, is a counter keeping a running total. It keeps the sum of the numbers in a series (the numbers 1 through 100) until the total reaches a particular value (15). Table 5-1 shows you how the values of both these variables change on the first five passes through the FOR\NEXT loop.

```
FOR Iteration = 1 TO 100
  Sum = Sum + Iteration
NEXT Iteration
```

Keep this totaler technique in the back of your mind; you'll need it to handle (ominous background music here) The Challenge later in this session.



**Table 5-1** Variables Used to Hold Running Totals

Iteration	'Old' Sum	Arithmetic	'New' Sum
1	0	$0 + 1$	1
2	1	$1 + 2$	3
3	3	$3 + 3$	6
4	6	$6 + 4$	10
5	10	$10 + 5$	15

'Old' Sum refers to Sum at the right of = in the second line of the program; 'New' Sum is sum to the left of =

### Nesting: Loops Within Loops

Loops can be, and often are, nested. To **nest** a loop means to include it in another loop. (It's possible to nest other control structures, such as IF..THEN statements, as well.) Here's an example using FOR\NEXT:



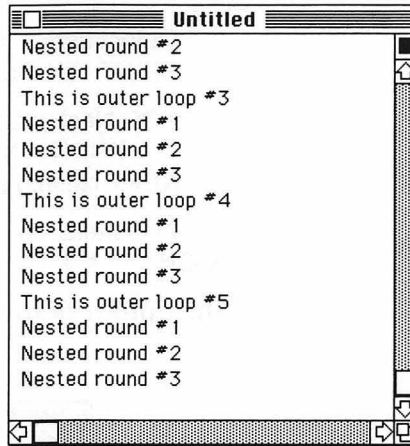
#### Do This

Type in and run the following program:

```
FOR Outer.Loop = 1 TO 5
  PRINT "This is outer loop #"; Outer.Loop
  FOR Nested.Loop = 1 TO 3
    PRINT "  Nested round #"; Nested.Loop
  NEXT Nested.Loop
NEXT Outer.Loop
```

Inner loops must always finish looping before outer loops. The inner-most loop always finishes first.

The inner loop must be completely finished with all three of its rounds before the outer loop can start its second round. The output of this program is shown in Figure 5-3.



**Figure 5-3** Nested DO\LOOPs

This next example uses nested DO\LOOP constructs to do exactly the same thing as the previous program. It too must finish all rounds of its inner loop before the outer loop's counter can increment.

```
DO
    Outer.Loop = Outer.Loop + 1
    PRINT "This is outer loop #"; Outer.Loop
    Nested.Loop = 0
    DO
        Nested.Loop = Nested.Loop + 1
        PRINT "Nested round #"; Nested.Loop
        IF Nested.Loop = 3 THEN EXIT
    LOOP
    IF Outer.Loop = 5 THEN EXIT
LOOP
```

Each EXIT works only for its own loop. When it's time for the nested loop's EXIT to work, BASIC branches to the statement following the first LOOP; that happens to be the IF line for the outer loop.

Here's a combination of the two types, using DO\LOOP for the outer loop and FOR\NEXT for the inner (nested) loop. Once again, the inner loop must go through all three iterations before the outer loop does its second iteration.

```

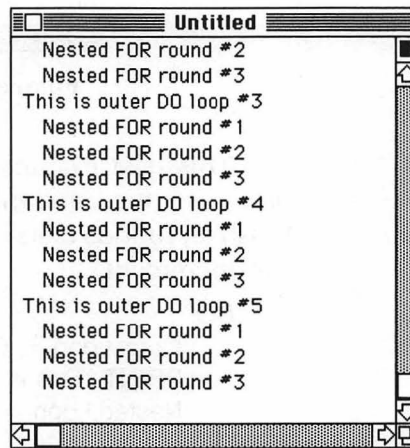
DO
    Outer.Loop = Outer.Loop + 1
    PRINT "This is outer DO loop #"; Outer.Loop

    FOR Nested.Loop = 1 TO 3
        PRINT "Nested FOR round #"; Nested.Loop
    NEXT Nested.Loop

    IF Outer.Loop = 5 THEN EXIT
LOOP

```

The output of this program is shown in Figure 5-4.



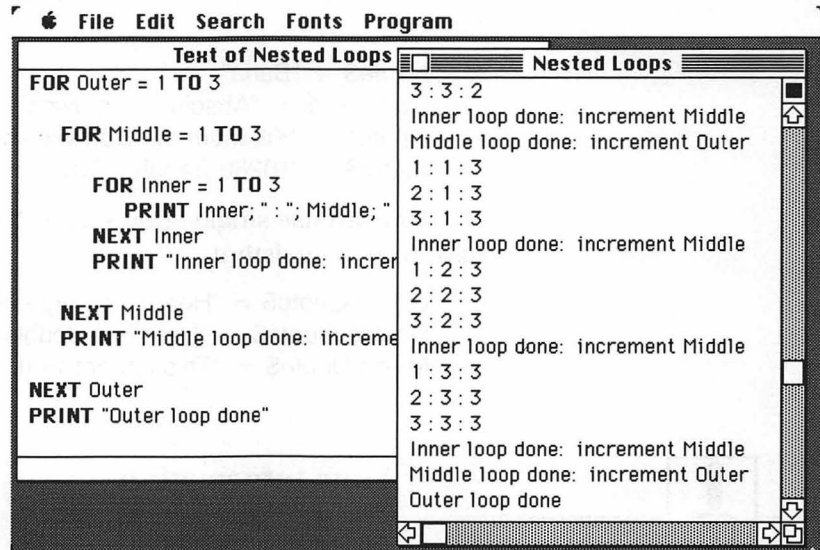
**Figure 5-4** Combination nested loops

You can nest loops as deeply as you want: you can have a loop within a loop within a loop within a loop and so on. Here's a final example, nesting FOR\NEXT loops three deep. Its listing and output are shown in Figure 5-5.

```

FOR Outer = 1 TO 3
  FOR Middle = 1 TO 3
    FOR Inner = 1 TO 3
      PRINT Inner; " : "; Middle; " : "; Outer
    NEXT Inner
    PRINT "Inner loop done: increment Middle"
  NEXT Middle
  PRINT "Middle loop done: increment Outer"
NEXT Outer
PRINT "Outer loop done"

```



**Figure 5-5** Output of triple-nested loops

The odometer on your car works on the nested loop principle; the tenths position has to increment 10 times before the units position can increment, the units position must increment 10 times before the tens position can increment, and so on.

Spend some time now playing with nested loops, using both DO\LOOP and FOR\NEXT structures. Construct the odometer I just talked about. Include some INPUT statements (along with clear prompting messages) in your experimenting, just for the practice. Take all the time you need. Then come back here and learn about strings.



## First Taste of String Variables

String variable names always end with the symbol \$.

Up to now you've used only numeric variables—variables capable of accepting numeric values. A **string variable** is a variable capable of representing any series (that is, any **string**) of text characters—numbers, letters, special symbols like & and ], or combinations of the three. String variable names look just like numeric variable names, except that string variable names always end in the dollar sign (\$).

### Setting Up String Variables

You assign values to string variables the same way you assign values to numeric variables, except that the string is usually in quotes:

```
Name$ = "Bana"  
Condition$ = "Absolute confusion"  
Quote$ = "Kill them all--God will know his own."  
Street$ = "10260 Bandleby Drive"
```

You can use single quotes as well as double quotes, but you have to be consistent:

```
Single.Quote$ = 'Here is a single quote'  
Double.Quote$ = "Here is a double quote"  
Mixed.Quote$ = "This cannot work"
```



### For Your Information

**Apostrophes Can Confuse BASIC** Be careful when using single quotes around a string that has an apostrophe in it—BASIC can get confused:

```
Confusing$ = 'This isn't gonna work, either.'
```

BASIC thinks the string is supposed to be *This isn*; it sees the rest of the sentence as some kind of mistake. To fix it, you can use two apostrophes in *isn't*. I know it looks funny, but life is like that sometimes.

## Section on Variable Redundancy Section

Just as with numeric variables, you can set one string variable to hold the same value as another. When you do that, you don't use quotes:

```
Book.Name$ = "Tutorial"    ! set value for a string variable
                           !   using quotes
```

```
Tome$ = Book.Name$        ! have a second string variable
                           !   hold same value as first
                           !   --don't use quotes here
```

```
PRINT Tome$               ! prints Tutorial
```

The original string variable still has its value:

```
PRINT Book.Name$          ! prints Tutorial
```

## INPUT with Strings

You use string variables for INPUT the same way you use numeric variables. The person answering the INPUT prompt just types in whatever response is appropriate, without using quotes. BASIC supplies the quotes, as shown in Figure 5-6.

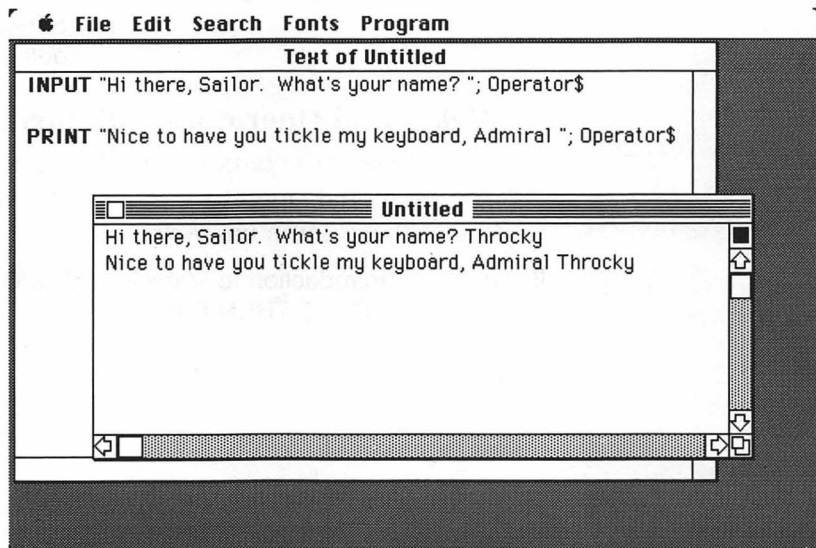


Figure 5-6 INPUT with string variable

You can't mix data types.

### How *Not* to Assign a String Value

You can't mix **data types**—that is, you can't assign a numeric value to a string variable, and you can't assign a string value to a numeric variable. All of the following examples get error messages:

This.String\$ = Numeric.Variable	! numeric to string variable
This.Number = String.Variable\$	! string to numeric variable
Value\$ = 27.45	! decimal numeric to string variable
Value = "27.45"	! string value to numeric variable
Price = \$7.49	! illegal string and numeric mix ! (\$ is a string character)

A number with quotes around it is no longer a number.

### Identity Crisis: Numbers in Quotes

As far as BASIC is concerned, once you've enclosed a number in quotes, it ceases to be a number and becomes a string. So the following forms are OK:

Value\$ = "27.45"	! quoted string to string variable
Value = 27.45	! numeric value to numeric variable
Price\$ = "\$7.49"	! quoted string to string variable ! (\$ meaning <i>string</i> and \$ meaning ! <i>dollar</i> is a coincidence)

### Relational Operators with Strings

Just as you can compare numbers and numeric variables using the relational operators (<, =, >), you can compare strings using these same characters:

```
IF Book$ = "Introduction to Macintosh BASIC" THEN PRINT "Fabulous!"
IF Steak$ <> Rare$ THEN EXIT
```

At first, the idea of a string “having the same value as” another string might seem a bit odd. When a string is said to hold the same value as another string, it means that the first string has exactly the same characters in exactly the same order as the second string. If there’s a space between the final letter and ending quote in the first string, then that space has to be there in the second string. And if the third character in the first string is an uppercase Z, then the third character in the other string must also be a capital Z—a lowercase z won’t do! See Table 5-2 for a list of examples. None of the strings in the first column of Table 5-2 matches the corresponding string in the second column. Where variable names appear, assume that all have been assigned different values.

**Table 5-2** Non-Matching Strings

First String	Second String	Reason For Nonmatch
"Bookstore "	"Bookstore"	First string has a space
"Edge Canyon"	"Edge canyon"	"C" and "c" don't match
"Rat.tails"	"Rat tails"	"." and " " don't match
" Summer"	"Summer"	Leading space in first string
Animal\$	Animals\$	Second string variable ends in s
Note.Pad\$	Note/Pad\$	"." and "/" don't match

While the distinction between uppercase and lowercase letters is significant in quoted strings (“Hot Dog” and “hot dog” don’t match), it doesn’t matter in string variable names (Computerjunkie\$ and computerJunkie\$ represent the same value). Non-alphabetic characters, however, must always match precisely.

In case you were wondering, strings can also be *less than* and *greater than* each other. Luckily, you don’t have to worry about that for another couple of sessions.



### A String Variable Idiosyncrasy: The Null String

The **null string** is a string with no characters in it (bear with me). There are two basic ways to create a null string. The first is through an assignment statement:

```
Null$ = ""
```

The second is by just pressing Return without typing any other characters in response to an INPUT prompt. (Of course, the INPUT variable must be a string variable).

You'll discover a number of uses for the null string as you program more, but one way you can use it right away is as a signaling device. Consider the following example (be careful of those single quotes in the final INPUT line):



#### Do This

Type in and run this program:

```
DO
  PRINT
  INPUT "First number: "; First.number
  INPUT "Second number: "; Second.number
  PRINT "The sum is "; First.number + Second.number
  PRINT
  INPUT 'Enter "s" to stop or press Return to continue '; Signal$
  IF Signal$ = "s" THEN EXIT
LOOP
PRINT "Thank you."
```

The program prompts for two numbers, adds them together, displays the results, and then waits for you to tell it what to do next. If you enter an "s" the program stops. But if you just press Return, assigning the null string to Signal\$ the program loops back for another round of adding.

You could use a numeric variable in this kind of situation, telling the operator to enter a 0 to stop or a 1 to continue, but that seems kind of unnatural. It's a lot more human to think of "s" to stand for *stop*, and it's easier just to use the Return key to mean *do another round*.



## Pop Quiz

### Question 3

What happens if the computer operator types "S" instead of "s"? What happens if the computer operator types *aardvark*? You'll find the answers at the end of the session.

Using the null string in this kind of situation can also save the operator some finger fatigue if there's a lot of adding to do. Some poor soul might have to add together hundreds of sets of numbers. Extra keystrokes add up fast!



## For Your Information

**Danger: Watch Your Variable Types** If you're going to accept a null string as a response to an INPUT, you *must* use a string variable. If you try to use a numeric variable, you'll get the error message "Expected a number;" the program will stop, and all your friends will point at you and laugh.

## Experiment Time!

Take some time now to experiment with strings. Use them with INPUT and PRINT statements. Move them to different positions in nested loops and see what happens. Rewrite some of the programs from this session using only variables in PRINT statements; assign all the sentences and phrases to string variables at the beginning of the program, like this:

```
Sentence$ = "This is iteration #"  
FOR Sample = 1 TO 5  
    PRINT Sentence$; Sample  
NEXT Sample
```

After you've done that for a while, go on to finish this session—and test how much you *really* understand with The Challenge!

---

### The Challenge: Program a BASIC Cash Register

---

you're going to create a program similar to one that operates every time you go to a supermarket. To write this program, you'll need most of the keywords you've learned so far (there's a list of them at the end of the session), plus one you haven't learned yet—CLEARWINDOW—which is explained a couple of paragraphs down. I'll start you off, then you're on your own.

Here's the scenario: It's 7:30 P.M. on a Friday evening. Half the known universe is coming to a party you're throwing tonight, and it's due to start at 9:00. You realize you're totally out of Crunchos, avocado dip, and beer nuts. You rush to the supermarket, pick up the supplies, and join the end of the Express Nine-Items-or-Less line, secretly praying that the six-pack counts as one item. Suddenly the cashier screams—the automated register (which happens to be driven by a Macintosh) is down! Its program has a bug! The distraught cashier cries “Is there a programmer in the house?” You haven't got much time, but your duty is clear.

#### Planning Phases

Before you reprogram the supermarket's Express Macintosh, you need to plan out the code. Your cash register program must do four things: it must accept a series of prices for different items (no more than nine—this is an express line), total up the items, tell the cashier what to charge the customer, and then prepare to move on to the next customer. Think of it as a four-phase operation:

- Input phase—accept the prices of the items
- Process phase—add up the prices
- Output phase—report the total
- Reset phase—prepare for the next customer

Your program can combine operations (for instance, it might accept the price of an item and immediately add it to a running

total, thus combining the input and process phases), and it can swap operations (for instance, doing a reset operation first), but it is likely to need to do something for each of the four phases.

### **The CLEARWINDOW Keyword**

CLEARWINDOW is one of the simplest keywords you'll learn in Macintosh BASIC. It just clears out everything in the output window. Note that I say *everything* in the output window; that includes any material that might have scrolled out of sight. You use CLEARWINDOW to keep your output windows tidy and uncrowded. It adds clarity to your reports much as PRINT statements can be used to improve clarity by adding blank lines. CLEARWINDOW has no effect on the listing window. Here's an example of how to use this keyword (you don't have to type in the PRINT lines if you don't want to; just enter the whole DO\LOOP structure):

```
PRINT "Type in something and press Return."
PRINT "I'll echo whatever you type."
PRINT "If you enter 'stop' I'll end the program."
PRINT "If you press Return without typing anything,"
PRINT "I'll erase the screen."
DO
    INPUT Answer$
    IF Answer$ = 'stop' THEN EXIT
    IF Answer$ = "" THEN CLEARWINDOW ELSE PRINT Answer$
LOOP
PRINT "Now go on to answer The Challenge!"
```

### **The Challenge**

You have 45 minutes to reconstruct the program used to produce an output window like the one shown in Figure 5-7. My solution is at the end of the session—but don't look. Here are some hints:

- Each time the price of an item is entered, it is added to a running total.
- No more than nine items per customer are accepted.
- The window is always erased between customers.
- Before a new customer's items are entered, the previous running total is reset to zero.





Figure 5-7 Possible output of "The Challenge"

### Don't Let the Bugs Get You Down

The Challenge is not an easy assignment. Don't be discouraged if your program doesn't work the first time, or the second time, or the third. Just keep at it! Every time you fix a bug, you learn something about programming. Hang in there.

## Summary

### New Terms

**Control structure** a group of BASIC statements bounded by keywords and operating in a reliable and consistent manner.

**Data type** a specific kind of information, recognizable to BASIC as such by the symbol appearing (or *not* appearing) at the end of a variable name. You know about two data types: strings and numeric.

**Loop variable** variable whose value changes on each pass through a loop. Some people call it the **index variable**.

**Nest** to enclose one within another.

**Nested loops** loops wholly enclosed within other loops.

**Null string** a string containing no characters.

**Pass** a single loop iteration.

**String** sequence of text characters.

**String variable** variable ending in the special character \$ (called dollar or string), capable of representing any series of text characters.

**Syntactic outline** symbolic representation of the form of a BASIC construct or statement.

---

## Programming Statements and Characters

---

**\$** character affixed to the end of a string variable name.

**CLEARWINDOW** clear everything in the output window.

**FOR loop.var = start.val TO end.val STEP step.val \NEXT loop.var** establish a looping structure in which everything between the lines containing the keyword **FOR** and the keyword **NEXT** are repeated a specified number of times; variable *loop.var* increments in steps of *step.val*, starting with the value *start.val* and ending when the value of *loop.var* exceeds that of *end.val*.

---

## Pop Quiz Answers

---

### Question 1

A step value of 0 says "Add zero to the value of the loop variable after each completed pass." Since the value for the loop variable never gets beyond the starting value, it can't reach the finishing value and so loops forever. The only time this isn't true is when you give a starting value higher than the finishing value, which brings us to Answer #2.

### Question 2

BASIC checks starting and ending values at the start of the loop. If the starting value is greater than the ending value, the loop isn't executed. The pattern looks like this:

```

Loop.Variable = Starting.Value
Limit = Some.Value.Or.Other
DO
  IF Loop.Variable > Limit THEN EXIT
  Counter = Counter + 1
  PRINT "This is iteration #"; Counter
LOOP

```

By the way, before this example can work, you need to give numeric values to *Starting.Value* and *Some.Value.Or.Other*. How come?

### Question 3

The answer to both questions is the same: the program goes through the loop again. Remember that uppercase and lowercase "s" are two different animals when they're in strings. The code says

```
IF Signal$ = "s" THEN EXIT
```

As far as BASIC knows, there are only two realities to consider: *s* or *not s*. Whether you type "S," *aardvark*, your mother's maiden name, or just *Return*, the result will be the same.

### My Solution to The Challenge

Figure 5-8 shows my solution. If you came up with another solution that works, you get full credit (but who's counting?). Note that I reset the value of *Total* to zero after each customer. If I don't do that, each shopper will be charged not only for his or her own groceries but also for the groceries of every shopper who came before. This might strain customer relations.

---

**Commands, Menu Items, Keywords You Know So Far**


---

**Menu Commands and Editing Keys**

Backspace key	Paste
Clear	Run
Copy	Save
Cut	Select All
Halt	Tab key
New	Undo
Open	⌘ key

**Programming Characters**

+	;	-
\$	/	!
*	>	=
<		

**Programming Keywords**

DO\LOOP	IF...THEN
ELSE	INPUT
CLEARWINDOW	PRINT
EXIT	
FOR....TO...STEP\NEXT	

---

**Bughouse**


---

This program is supposed to loop backwards from 10 to 1, each time showing what iteration it is and then clearing the window. It has at least five bugs. Get the bug spray.

```

Words = "This is backloop #"
Start = 10
Finish = 3
Increment = 1
CLEARWINDOW
FOR Number = Start TO Finish STEP Increment
  PRINT Word$, Numbers
NEXT Number
  
```

---

**Listing for the Challenge**

```
DO
  CLEARWINDOW
  FOR Item = 1 TO 9
    PRINT "Price for #"; Item; ": $";
    INPUT ""; Price
    IF Price = 0 THEN EXIT
    Total = Total + Price
  NEXT Item
  PRINT
  PRINT "Total: $"; Total
  PRINT "Thanks for shopping at MacMart!"
  PRINT
  PRINT "Press Return for next customer"
  PRINT "Or press 's' and Return to stop: ";
  INPUT ""; Order$
  IF Order$ = "s" THEN EXIT
  Total = 0
LOOP
PRINT
PRINT "Register Closed"
```

---

**Figure 5-8** Listing of one solution to "The Challenge"



## SESSION

# 6

## Graphics and the Mouse

---

**O**ne of the Macintosh's strongest points is its graphics capability. The Macintosh's display screen is made up of around 200,000 individual dots, each of which can be turned on or off like a tiny light. This session teaches you the rudiments of turning those little lights on and off in meaningful ways.

In this session you'll experiment with a variety of graphic objects, including points, lines, rectangles, and ovals; and you'll learn how to outline and fill various shapes. This session doesn't tell you everything about graphics on the Macintosh; that would take a whole book. But there's enough here to get you started.

### The Graphics Area

---

When you run a program, BASIC creates an output window that's 241 dots wide by 241 dots high. The technical term for a dot is **pixel**, a corruption of an abbreviation for *picture element* (*pictel* became *pixel* somehow—probably because it's easier to say). You can change the size of this **graphics area** by manipulating the size box; if the output window seems too small (which it often will), just make it larger.

Think of the window as containing invisible columns and rows. You can refer to any dot in the window by giving the dot's horizontal (column) and vertical (row) **coordinates**. For example, the dot at the upper left corner of an output window is at coordinate 0, 0 (column 0, row 0), and the one at the lower right is at coordinate 240, 240 (column 240, row 240). Each pair of coordinates that specifies a dot is called a **coordinate set**.

## The PLOT Statement

To turn on individual dots you use the **PLOT** statement.



### Do This

Type in the following program; running it turns on the dots at the center and near the boundaries of the output window. Be sure to make the output window larger so that it fills most of the screen

```
PLOT 10, 10      ! upper left
PLOT 10, 230     ! lower left
PLOT 230, 230    ! lower right
PLOT 230, 10     ! upper right
PLOT 115, 115    ! center
```

Figure 6-1 shows what the program produces.

You need two numbers separated by a comma for each PLOT statement.

Imagine an invisible pen to go with the invisible columns and rows. When you say PLOT, the pen moves to the specified row and column, presses down on the glass paper to make a dot (turn on a pixel), and then lifts off the paper and waits for the next instruction.

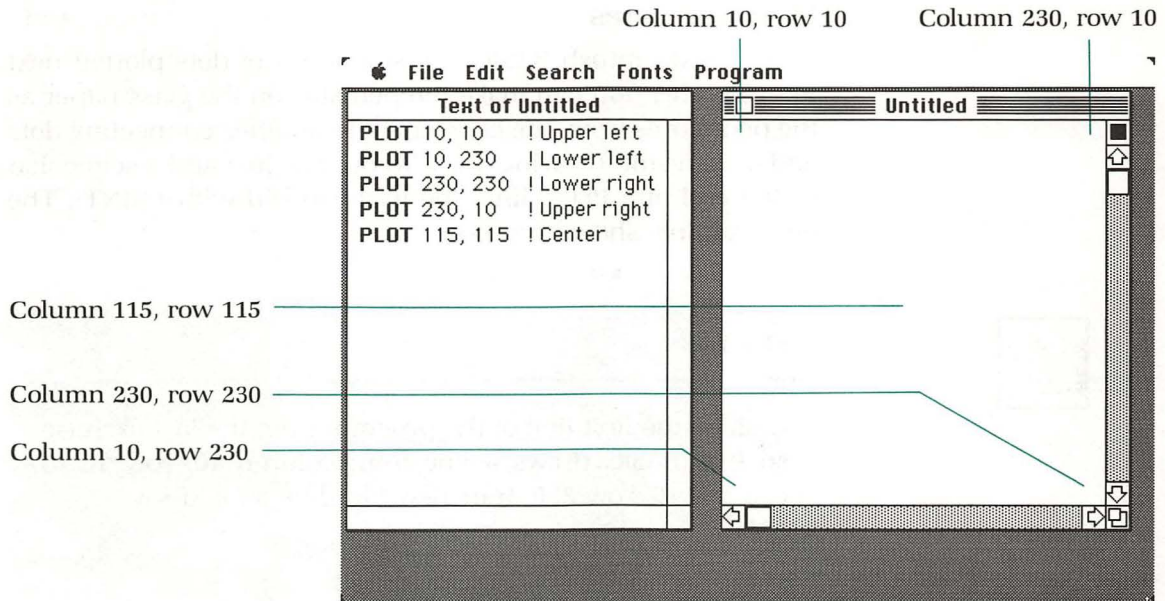


Figure 6-1 Simple PLOT program



### For Your Information

**About Turning Dots On and Off** When I say “turn on a dot,” I really mean “reverse the normal state of a dot.” Macintosh BASIC’s output window is preset to be all white, like a blank sheet of paper waiting for you to draw on. So if you want to get technical, each time you plot a point you’re turning a dot *off*, since they’re all lit up to begin with. But most people with backgrounds in computer graphics think of plotting a point as turning on a dot or pixel, since most computers have all-black output windows. Try not to think about it.



## Drawing Lines

A line in Macintosh BASIC is just a series of dots plotted next to each other. You can make the pen stay on the glass paper as the pen moves from one coordinate to another, connecting dots and (as a result) drawing lines. To do this, just add a semicolon to the end of a PLOT line (the way you did with PRINT). The next example shows you how.



### Do This

Change the first line of the program from the last exercise so that BASIC draws a line from column 10, row 10 to column 10, row 230. Your new first line should say

```
PLOT 10, 10;
```

Instead of having five dots, you'll now get a line and three dots. The dots at the upper left and lower left now connect; all the dots in column 10 between rows 10 and 230 are turned on. Figure 6-2 shows the line you'll get.

If you put a semicolon at the end of each line, nearly all the dots will connect.



### Do This

Change the program so that there's a semicolon at the end of the first four PLOT lines (you don't need one at the end of the fifth line) and then execute the program.

```
PLOT 10, 10;      ! upper left
PLOT 10, 230;     ! to lower left
PLOT 230, 230;    ! to lower right
PLOT 230, 10;     ! to upper right
PLOT 115, 115     ! to center
```

Figure 6-3 shows what you should get.



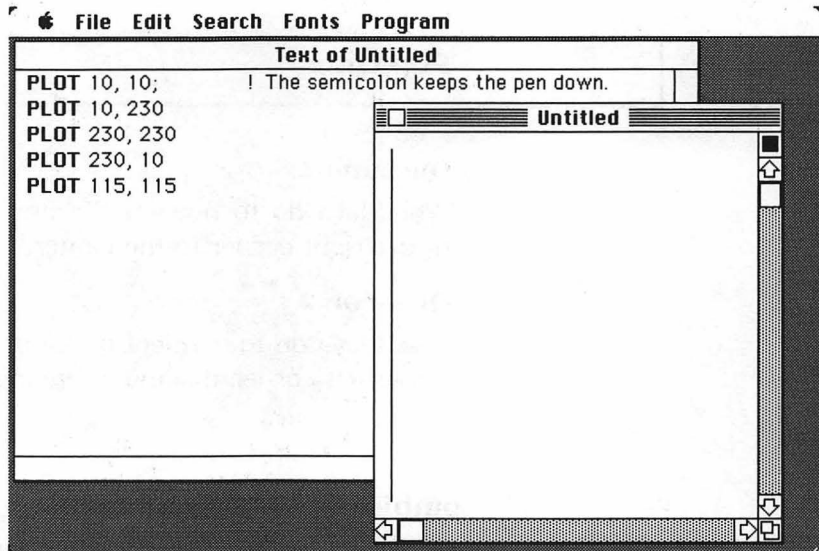


Figure 6-2 Keeping the Pen down with a semicolon

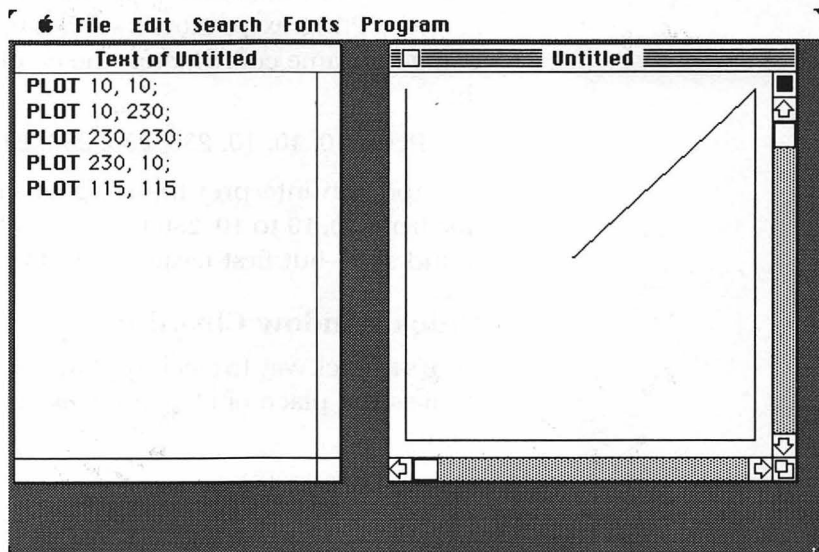


Figure 6-3 Drawing more lines



---

## Pop Quiz

### Question 1

Whaddaya do to prevent the line being drawn from the upper right corner to the center?

### Question 2

Whaddaya do to connect the upper right corner with the upper left corner, making a rectangle?

## Combining PLOT Statements

BASIC has a built-in shortcut that lets you write the program you just wrote in one line. Because the semicolon at the end of each statement means “leave the pen down” and each successive line is a PLOT statement, you can leave out all instance of the keyword PLOT except the first one and combine the statements. Using the same coordinates, the combined statement looks like this:

```
PLOT 10, 10: 10, 230; 230, 230; 230, 10; 115, 115
```

You can interpret the code as saying “Draw a continuous line from 10, 10 to 10, 230 to 230, 230 to 230, 10 to 115, 115.” Try it and see—but first clear out and reuse the listing window.

## Quick Window Cleaning

Here's a quick way to clear the window with just three keystrokes; it takes the place of choosing Select All and Cut from the Edit menu.



### Do This

Clear out the listing window for reuse, then enter the single line of code listed above.

1. Click the Close box on the output window, bringing the listing window to the front.
2. Hold down the  $\mathbb{H}$  key.
3. Press and release the “a” key to select the entire window.
4. Press and release the “x” key to cut all the selected text.
5. Release the  $\mathbb{H}$  key.
6. Type in this code:

```
PLOT 10, 10; 10, 230; 230, 230; 230, 10; 115, 115
```

7. Run the program.

This procedure is like a `CLEARWINDOW` for the listing window.



### For Your Information

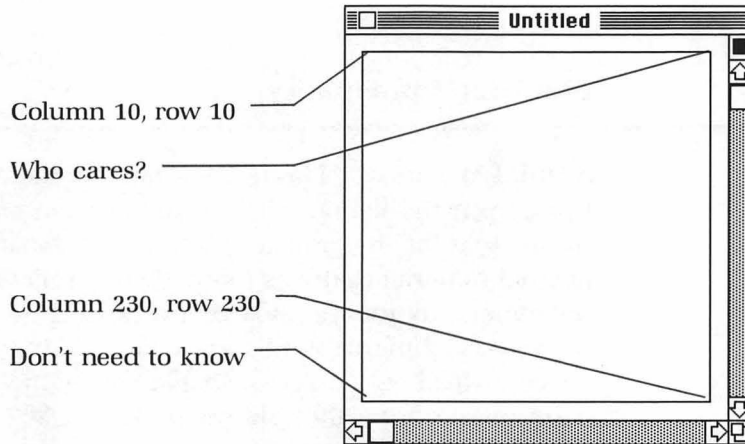
**A Quick Warning** This quick-clean technique *cuts* everything from the listing window rather than *clears* it; that means that the material goes into the Clipboard, replacing any old material that was there. If you don't want to wipe out what's in the Clipboard, you can still use  $\mathbb{H}A$  to select everything in the listing window—but press Backspace instead of “x” to *clear* instead of *cut*. If you mess up, choose Undo from the Edit menu (or use  $\mathbb{H}Z$ ).

## From Points to Rectangles

A few paragraphs earlier you got a pop quiz in which you were asked to complete a rectangle. Using the knowledge you had then, you had to type a total of five sets of coordinates to make the rectangle: upper left (1) to lower left (2), lower left to lower right (3), lower right to upper right (4), and upper right back to upper left (5). Using the keywords **FRAME** and **RECT**, however, you can draw the rectangle with just two coordinates—the upper left and the lower right. Clear the windows; then type and execute this line:

```
FRAME RECT 10, 10; 230, 230
```

The keyword **FRAME** means “draw a boundary around”; the keyword **RECT** is short for *rectangle*. So the code says “Draw a boundary around a rectangle whose upper left corner is at column 10, row 10 and whose lower right corner is at column 230, row 230.” The rectangle this produces is shown in Figure 6-4.



**Figure 6-4** Drawing a rectangle with **FRAME RECT**





### For Your Information

**I Lied** Actually, there is a slight difference between rectangles you crunch out with PLOT statements and those you make using FRAME RECT. When you use words like FRAME, BASIC draws the shape on the screen just *within* the coordinates you give. To match the PLOTted rectangle exactly, you'd have to add 1 to each of the numbers describing the lower right coordinate—in other words, make the line read

```
FRAME RECT 10, 10; 231, 231
```

### Shapes Need Two Keywords

When you plot a point or a series of points, you need only one keyword—PLOT. But when you draw a shape such as a rectangle, you need *two* keywords. The first keyword says what you want to do with the shape, and the second one tells BASIC what the shape is. So far you know how to do only one thing with a shape—FRAME it—and how to make one shape, RECT. Later there'll be more.



### Pop Quiz

**Question 3** What line of code would you add to draw a diagonal line from the upper left corner to the lower right corner of the rectangle?

### Using Variables with Graphic Objects

So far you've been drawing points, lines, and rectangles using numeric constants. In this section you'll see how to use variables in your drawing.

Beginning with the coordinates you just used, substitute variables to draw a rectangle with upper left coordinate 10, 10 and lower right coordinate 230, 230. Here's what the variables might look like:

```
Left.Column = 10  
Top.Row = 10  
Right.Column = 231  
Bottom.Row = 231
```

The original code looked like this:

```
FRAME RECT 10, 10; 231, 231
```

Replacing the numeric constants with variables, you get:

```
FRAME RECT Left.Column, Top.Row; Right.Column, Bottom.Row
```



### Do This

Type in and run this program:

```
Left.Column = 10  
Top.Row = 10  
Right.Column = 231  
Bottom.Row = 231  
FRAME RECT Left.Column, Top.Row; Right.Column, Bottom.Row
```

You get exactly what you got before; a rectangle taking up most of the output window. The only difference is in the code. Variables add a great deal of flexibility to programming, as you'll see as soon as you do the next two modifications. First, modify the program to see what direction you're heading in.



### Do This

Add a new line of code to the program, as shown below, and then execute it:

```
Left.Column = 10
```

```
Top.Row = 10
```

```
Right.Column = 231
```

```
Bottom.Row = 231
```

```
FRAME RECT Left.Column, Top.Row; Right.Column, Bottom.Row
```

```
FRAME RECT Left.Column + 5, Top.Row + 5; Right.Column - 5, Bottom.Row - 5
```

You get a rectangle within a rectangle. The inner rectangle has been pulled in five dots' worth on all four sides.

Of course, what you can do once with an added line of code, you can do many times with a loop.



### Do This

Modify the program so that it produces rectangles within rectangles within rectangles. . . .

```
Left.Column = 10
```

```
Top.Row = 10
```

```
Right.Column = 231
```

```
Bottom.Row = 231
```

```
FOR C = 5 TO 100 STEP 5
```

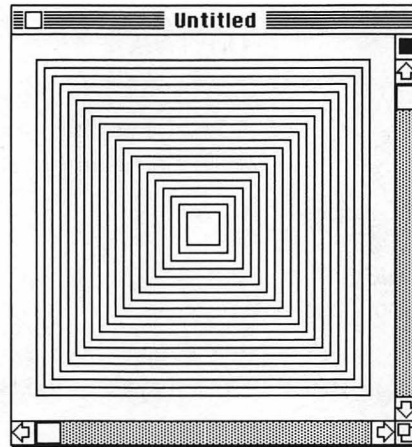
```
  FRAME RECT Left.Column + C, Top.Row + C; Right.Column - C, Bottom.Row - C
```

```
NEXT C
```

Predict what this program will do *in each line* before you execute it.

Each time through the loop, the loop variable C (for Change) increases by 5, and the rectangle gets pulled in five dots on each side. The result is shown in Figure 6-5.





**Figure 6-5** Rectangles within rectangles within rectangles...

### Slowing Things Down

If this program went by too fast for you to follow, you can use the BASIC keyword **BTNWAIT** to slow it down. Insert **BTNWAIT** on its own line, just before the **NEXT C** line at the bottom of the loop, as shown below. **BTNWAIT** makes the computer wait until you push the mouse button before going on.

```
Left.Column = 10
Top.Row = 10
Right.Column = 231
Bottom.Row = 231

FOR C = 5 TO 100 STEP 5
  FRAME RECT Left.Column + C; Top.Row + C; Right.Column - C, Bottom.Row - C
  BTNWAIT      ! Here's the new keyword
NEXT C
```

After you type in **BTNWAIT**, execute the program again. You'll find that now, BASIC draws a new rectangle within the last one each time you press the mouse button.

Store this program on a disk under the name *Rectangles* (creative name, eh?) and then spend some time playing with **PLOT** and **FRAME RECT**, using variables and loops. After that, go on to learn a few more handy keywords that use the mouse.



---

## Some Mouse Words

---

Much of what happens on the Macintosh is or can be controlled with the mouse. So BASIC has a number of keywords relating directly to the mouse. You've just learned one of the words, **BTNWAIT**; it suspends the operation of a BASIC program until you press the mouse button.

Several mouse keywords are actually **functions**. Functions are formulas and minor programs built into BASIC that manipulate numbers and other information in highly specialized ways. The two you'll learn here tell you where the mouse pointer is and whether somebody is pushing the mouse button. These numeric **system functions** act like variables except that, instead of your giving a value to the computer, the computer gives the value to you.

### The Pointer Position

Knowing the pointer position involves knowing both the column and the row where the pointer is. It's like plotting a point in reverse. To plot a point, you tell the computer the horizontal and vertical coordinates; using the pointer position functions, the computer tells *you* the pointer's column and row. The system numeric function **MOUSEH** (H stands for *horizontal*) gives you the current pointer column; **MOUSEV** (V for *vertical*) gives you the current pointer row.

### Finding the Column

First, see for yourself how **MOUSEH** works.



### Do This

Write a short program to show how MOUSEH works.

**1.** Type in this program:

```
DO                ! Start the loop here
  BTNWAIT         ! Wait until the button's pushed
  PRINT MOUSEH    ! Tell what column the pointer's in
LOOP              ! Go back to the top of the loop
```

**2.** Run the program.

**3.** Set up the output window so that it fills most, but not all, of the screen; leave borders all around it.

**4.** Move the pointer just to the right of the left edge of the output window and press the mouse button.

**5.** Move the pointer just to the left of the right edge of the output window and press the button.

When I did the preceding exercise, I got 1 for my first number and 383 for my second; that is, the first time I pressed the mouse button, the pointer was pointing to column 1, and the second time it was pointing to column 383. Your numbers may be slightly different.

Spend a few minutes moving the pointer around and getting more column positions. At some point during your mouseplay, move the pointer to about halfway between the left and right sides of the output window; you should get a number close to 200.

### About Out-of-Bounds Numbers

In the exercise coming up, you'll see that MOUSEH sometimes produces minus numbers or numbers too large to represent the area bounded by the output window's sides. That's because the output window is only part of the Macintosh's graphics area. To learn more about this, you'll need to modify the program you've been using.



### Do This

---

1. Deactivate the `BTNWAIT` statement by preceding it with a comment marker character (!); then run the program again.
2. Move the pointer until it bumps up against the left of the screen, outside the output window. Then do the same thing with the far right edge of the screen.
3. Use `⌘ H` to stop the program.



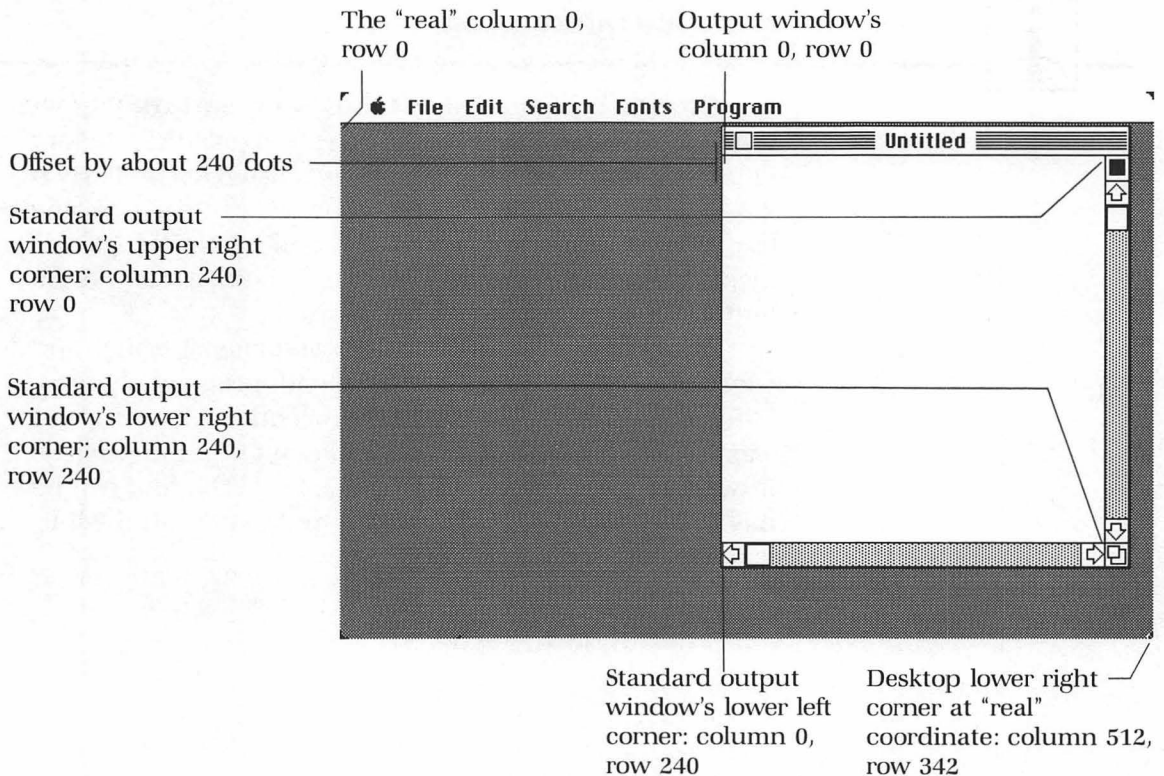
### For Your Information

---

**Use ! to Store “Temporary” Lines** You have to deactivate the `BTNWAIT` statement because a running BASIC program ignores button presses for statements like `BTNWAIT` when the pointer is outside the active output window (pressing the mouse button still activates system-level things like menu items and close boxes, however). BASIC *waits* for a button press, but won't recognize it!

As you learn more and more statements, you'll find yourself experimenting more often with program changes. When you want to eliminate a line of code temporarily but want to keep it around so that you don't forget it, precede it with the comment marker character `!`. The line will be there when you need it, but while you don't need the line, the computer will ignore it.

The numbers that appear in the output window are outside the bounds I gave for this window earlier. That's because the output window that BASIC gives you takes up only part of the whole graphics area, which is actually as wide as the screen. You get a minus number when you move the pointer to the left edge of the screen because the output window is offset from the left edge by that many columns. Since you can move the output window around, the minus numbers you get will vary depending on the position of the output window on the desktop (remember that the upper left corner of the output window has coordinates 0,0 no matter where you move the window). The entire visible graphics area is actually about 500 dots wide, as shown in Figure 6-6.



**Figure 6-6** Local coordinates of output window





### Do This

Make the output window as wide as the screen and then take “extreme” readings again.

1. Move the output window to the left edge of the screen by dragging the title bar to the left.
2. Drag the size box to the right edge of the screen.
3. Choose Go from the Program menu to make the halted program continue (or use ⌘ G).
4. Move the pointer to the left edge.
5. Move the pointer to the right edge.

The first column, column 0, is always the one at the extreme left edge of the output window, no matter where you position the output window on the screen. As you’ve already seen, you can tell when the pointer is to the left of the output window because you get a minus reading with MOUSEH. This last fact will come in handy later.

### Finding the Row

So much for columns. To find out about rows, you’ll need to change the program slightly again.



### Do This

Get rid of the oversized output window, take out the exclamation mark in front of BTNWAIT, change MOUSEH to MOUSEV, and execute the program again. Now the number you get each time you press the mouse button tells what row you’re pointing to—the vertical position of the pointer. Try it with the pointer at the top of the output window and then with the pointer at the bottom. When I do it, I get 0 and 240.

### Exceeding Vertical Boundaries

Again, if you move the pointer above the output window (assuming the `BTNWAIT` statement is deactivated), you'll get a minus number; move it below the output window and you'll get a higher number. I get  $-39$  above the window and  $300$  when I go to the bottom of the screen.

Try doing vertically what you did horizontally before; that is, make the output window fill as much of the vertical part of the screen as you can. You can't move the window all the way to the top of the screen, however; you can move it up only as high as the menu bar. The menu and title bars are each about 20 dots high, so the smallest negative number you can get with `MOUSEV` is about  $-40$ .

Going the other way is a different story. You'll soon get more numbers in the output window than can show; as you expect, the numbers at the top of the window scroll out of sight to make room for the new ones at the bottom. Meanwhile, look at the value of the numbers—they'll increase beyond the  $300$  "maximum" row number. The output window has a huge potential number of rows—many hundreds—that you don't see unless you ask to. However, for most practical applications in beginning programming, you'll need only the first 300 rows.

As with earlier long outputs, the old numbers have scrolled out of sight but aren't forgotten. If you click the arrow at the top of the scroll bar, you'll see the old numbers reappear. The original number is still 1 or 0 or some other low number depending on how close you were to the actual top of the output window.

### State of the Mouse Button

The third mouse function is `MOUSEB`. `MOUSEB` can report only two numbers: 0 if the button on the mouse is not being pressed, 1 if it is being pressed.



## Do This

Change the program you've been working with to show the position of the pointer.

1. Get rid of the output window by clicking the close box.
2. Clear the listing window using ⌘ A ⌘ X.
3. Enter this program:

```
DO
  IF MOUSEB = 1 THEN PRINT MOUSEH; " "; MOUSEV
LOOP
```

4. Predict what the program will do; then run it.
5. Move the mouse around and occasionally press the button. Do this for nine or ten presses.
6. Finally, hold the button down and roll the mouse around.



## Pop Quiz

### Question 4

What happens if you change the program to say IF MOUSEB = 0 instead of IF MOUSEB = 1? Predict it first—then change the program to test your answer.



## For Your Information

**System Function Names Can't Be Variable Names** I said earlier that system functions are like variables. But only the system can assign values to system functions; they are all keywords, reserved for the computer's use. So you can't use them as variable names.

## Play Time #1

Now you've got enough information and graphics keywords to do some pretty interesting things on the screen. I'll start you off with a few sample programs—but the real learning comes when you experiment on your own.



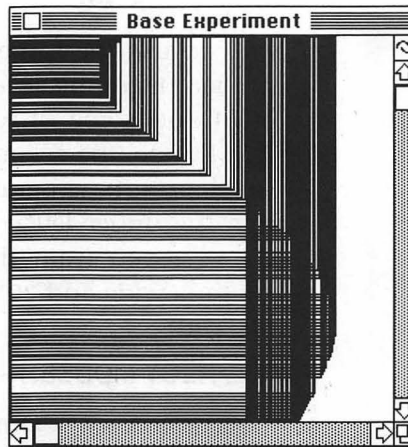
### Do This

1. Type in the following program.
2. Play computer—figure out from the keywords what actions you, as the computer operator, will have to take when you execute the program.
3. Run it.

```
DO
  FRAME RECT 0, 0; MOUSEH, MOUSEV
  IF MOUSEV < 0 THEN CLEARWINDOW
  IF MOUSEB = 1 THEN EXIT
LOOP
PRINT "That's it!"
```

Figure 6-7 shows what you should get. Save this program under the name *Base Experiment*; then go on.





**Figure 6-7** Dynamic graphics using FRAME RECT and mouse

### For Those a Bit Lost

Here's a line-by-line explanation of the preceding program. If you are sure you already understand what every line is doing, you can skip down to the next section.

**DO** Start an endless loop.

**FRAME RECT 0,0; MOUSEH, MOUSEV** Outline the shape of a rectangle whose upper left corner coincides with the upper left corner of the output window and whose lower right corner is marked by the tip of the pointer.

**IF MOUSEV < 0 THEN CLEARWINDOW** If the BASIC system function **MOUSEV** returns a number less than 0, which it will if the pointer is positioned above the top of the output window, then erase the output window.

**IF MOUSEB = 1 THEN EXIT** If the mouse button is pressed, leave the loop.

**LOOP** Go back to the top of the loop, marked by the keyword **DO**, and repeat the loop.

**PRINT "That's it!"** Print a message indicating that the program has ended.

## Play Time #2

You can use the same program to experiment with a number of different graphics forms, some of which you've already learned about and some of which you haven't, just by changing the FRAME line (the second line of the program). For instance, you can change the program so that, rather than drawing rectangles from the upper left corner, it draws rectangles of a given size wherever you want them. Let's say you want a rectangle 20 columns wide and 20 columns high. This line of code will give it to you:

```
FRAME RECT MOUSEH - 20, MOUSEV - 20; MOUSEH, MOUSEV
```

The line says "Outline a rectangle whose upper left corner is at a point 20 dots to the left of and 20 dots above its lower right corner; its lower right corner is marked by the tip of the pointer."



### Do This

Try it. Here's what the whole program should look like.

```
DO
  FRAME RECT MOUSEH - 20, MOUSEV - 20; MOUSEH, MOUSEV
  IF MOUSEV < 0 THEN CLEARWINDOW
  IF MOUSEB = 1 THEN EXIT
LOOP
PRINT "That's it!"
```

Figure 6-8 shows what the output of this program, which I've called *Little Boxes*, looks like.

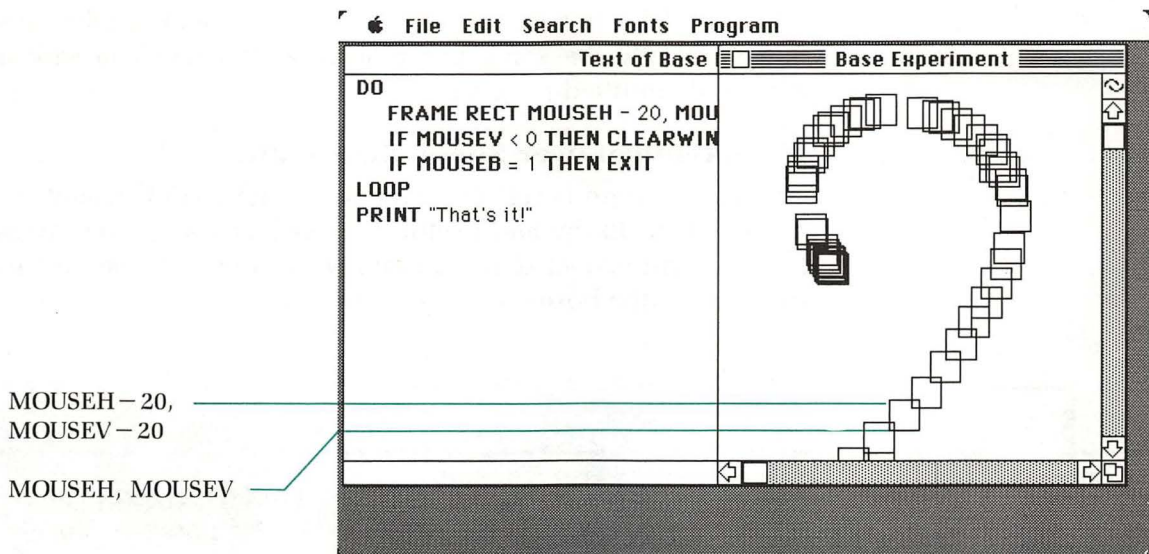


Figure 6-8 Little Boxes

### The Save a Copy In... Command

There's a command under the File menu you haven't used yet, although you first saw it in Session 3. **Save a Copy In...** lets you save different versions of the same program under different names. In this case, you'll save your new version of *Base Experiment* under the name *Little Boxes*.



### Do This

Save the program under the name *Little Boxes*.

1. Choose Save a Copy In from the File menu.
2. When the dialog box appears, type in the name *Little Boxes*.
3. Press the Return key.

This last step is a quicker alternative than clicking Save; both do the same thing. So does using the Enter key.



The dialog box for Save a Copy In should look familiar; it's almost exactly the same one that appears when you save a previously untitled program.

### More Control over Shape Placement

The last program is sort of flashy, but it isn't terribly useful. To make it both flashy *and* useful (for those of you in the Frank Lloyd Wright school of programming), you need to be able to put each of the boxes where you want it.



#### Do This

Change the program by adding the `BTNWAIT` keyword, but preserve the conflicting `MOUSEB` line for possible reuse later.

1. Add this line immediately after the `DO` line:

`BTNWAIT`

2. Place a comment marker character (!) before the `MOUSEB` line or delete the line entirely (using the symbol is preferred).
3. Figure out why step 2 is necessary.
4. Predict how to use the program and what it will do.
5. Run it.

Figure 6-9 shows the results.

### Why You Need to Inactivate IF MOUSEB

Of course, the easiest way to find out why you need to inactivate the `IF MOUSEB` line is *not* to inactivate it, execute the program, and see what happens. After you try that, read on.



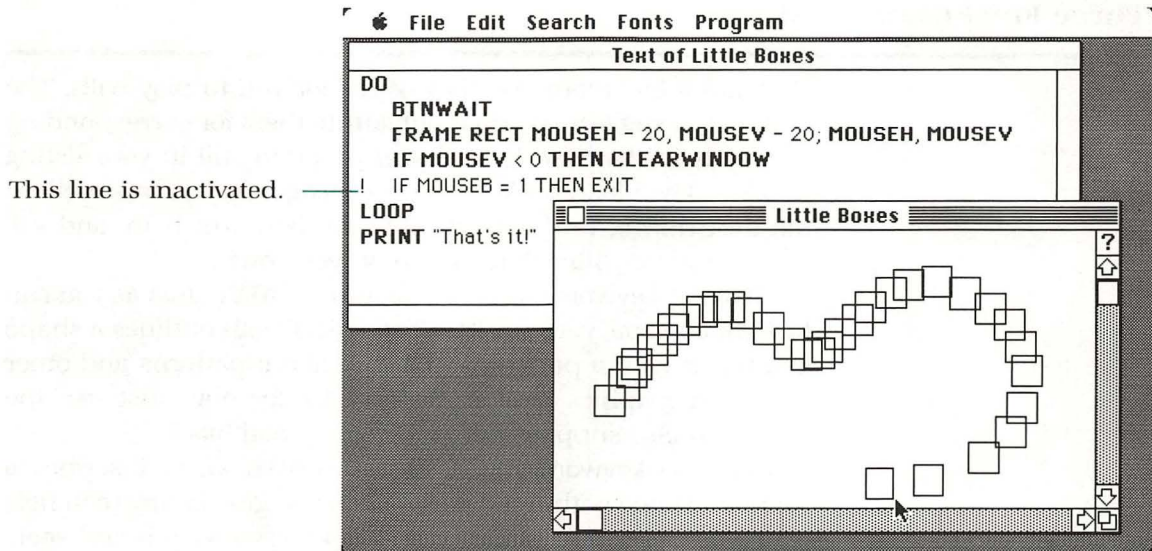


Figure 6-9 Little Boxes with BTNWAIT

You find that the program ends as soon as you draw your first square. Here's why: BASIC waits for you to press the button (BTNWAIT) before drawing the square. Next it checks to see whether you want the window cleared. Then—here comes the trouble—it tries to find out whether you want to leave the loop and end the program. If the button on the mouse is down, the program is over. Because the program is a lot faster than you are, BASIC will check the IF MOUSEB line long before you can release the button from the BTNWAIT line. (Don't feel bad. Your Macintosh will always be faster, but you'll always be smarter.)



## Pop Quiz

### Question 5

Assuming you first inactivate BTNWAIT, how can you use the MOUSEH function to tell the computer you want to leave the loop and end the program?

## Three Final Graphics Words

---

Here are a few more graphics words for you to play with. The first thing I suggest you do is substitute them for corresponding keywords in the *Base Experiment* program still in your listing window. Then load back the various programs you've stored on the disk during this session, plug the new words in, and see what happens. After that, you're on your own.

For the keyword **FRAME**, substitute **PAINT**. Just as you can **FRAME** a shape, you can **PAINT** it. **PAINT** both outlines a shape and fills it with a pattern. You'll read about patterns and other advanced graphics stuff in Session 10; for now, just use the pattern **BASIC** supplies automatically—solid black.

For the keyword **RECT**, plug in **OVAL**. **OVAL** inscribes a rounded shape within the parameters you give for any rectangle (so if you describe a square, you'll get circles—try it and see).

Finally, substitute the keyword **INVERT** for **FRAME**. I won't describe what **INVERT** does; I'll just give you a little program.

```
DO
    INVERT OVAL 0,0; MOUSEH, MOUSEV
LOOP
```

## Summary

---

### New Terms

---

**Coordinate** the point where a column and a row meet.

**Coordinate Set** the two numbers, first for column and second for row, describing a coordinate position. A comma separates the numbers.

**Function** formula built into **BASIC** that manipulates numbers or strings in some highly specialized way. Most pocket calculators, for example, have a built-in square root function.

**Pixel** the smallest picture element; a single dot on the Macintosh graphics area, one of about 200,000.

**System function** function that gives information about some changing state in the system, such as the current position of the mouse pointer.

---

## Menu Commands

---

**⌘ A** select entire listing window. Same as choosing Select All from the Edit menu.

**Go ( ⌘ G )** make a Halted program continue if the output window is active. If you use Go when a listing window is active, it has the same effect as choosing Run.

**Save a Copy In ( ⌘ I )** store a copy of the active listing window under the designated name. The program stored under the original name isn't changed.

---

## Programming Statements and Characters

---

**!** comment marker character, used in this session to deactivate a line of code without removing it from the program.

**BTWAIT** interrupt program execution until the mouse button is pressed.

**FRAME** draw a solid line around the shape whose upper left coordinate position is *left, top* and whose lower right coordinate position is *right, bottom*. A semicolon separates the coordinate sets.

**INVERT** reverse the state of all the dots in a given area whose upper left coordinate position is *left, top* and whose lower right coordinate position is *right, bottom*. A semicolon separates the coordinate sets.

**MOUSEB** system function that returns a 1 when the mouse button is being held down and a 0 when it is not down.

**MOUSEH** system function that returns a number in the range  $\pm 32767$ , representing the horizontal position of the pointer.

**MOUSEV** system function that returns a number in the range  $\pm 32767$ , representing the vertical position of the pointer.

**OVAL** BASIC graphics shape; a circular object inscribed within a rectangle whose upper left coordinate position is *left, top* and whose lower right coordinate position is *right, bottom*. A semicolon separates the coordinate sets.

**PAINT** fill with a specified pattern a specified shape whose upper left coordinate position is *left, top* and whose lower right coordinate position is *right, bottom*. A semicolon separates the coordinate sets.

**PLOT** light up (more correctly, turn off) one or a series of dots. PLOT takes a minimum of one coordinate set. If you use several sets, separate them with semicolons.

**RECT** BASIC graphics shape; a rectangular object whose upper left coordinate position is *left, top* and whose lower right coordinate position is *right, bottom*. A semicolon separates the coordinate sets.

## Pop Quiz Answers

### Question 1

Remove the semicolon from the end of the fourth line. A semicolon leaves the pen down, connecting the dots between the previously named point and the next one.

### Question 2

Add this new line between the fourth and fifth lines: PLOT 10, 10 Since the previous line ends in a semicolon, leaving the pen down (PLOT 230, 10;), all the dots between 230, 10 and 10, 10 will be lit.

### Question 3

The quickest solution is

PLOT 10, 10; 231, 231

### Question 4

You discover this answer by changing the program. Sorry.

### Question 5

I'd use MOUSEH the same way I used MOUSEV:

IF MOUSEH < 0 THEN EXIT

If you move the pointer beyond the left edge of the output window, MOUSEH will be less than 0, and BASIC will leave the loop.

## Commands, Menu Items, Keywords You Know So Far

### Commands and Menu Items

Backspace key	Paste
Clear	Run
Copy	Save
Cut	Select All
Halt	Undo
Open	⌘ key
New	

### Programming Characters

+	-	/
*	=	>
<	;	!
\$		

### Programming Statements

BTNWAIT	MOUSEB
CLEARWINDOW	MOUSEH
DO\LOOP	MOUSEV
ELSE	OVAL
EXIT	PAINT
FOR...TO...STEP\NEXT	PLOT
FRAME	PRINT
IF...THEN	RECT
INPUT	
INVERT	



---

**Bughouse**

---

I'm not going to tell you what this program is supposed to do; when you've got it all debugged, you'll see for yourself. I'll give you a hint: the program is called *Mirror*. It's got six bugs, all in keywords.

Maximum = 240

Change = 25

DO

    INVERT CIRCLE MOUSECOLUMN, MOUSEV; MOUSEH + Change, MOUSEV + Change

    NewMouseH = Maximum - MOUSEH

    NewMouseV = Maximum - MOUSEV

    INVERSE OVAL NewMouseH - Change, NewMouseV - Change; NewMouseH, NewMouseV

    IF MOUSEBUTTON = 1 THEN CLEARSCREEN

LOOP

# SESSION



## Random Subroutines

---

**T**he subroutine is one of the most important concepts in programming. It's made up of a group of program lines that you use often in a program but write only once; any time you want to use it, you just summon it by name. This session teaches you how to write and use subroutines in some very practical ways—including one that might help you get back the money you laid out for your Macintosh.

You'll also get your first taste of **numeric functions**, which are designed to manipulate numbers in different ways. Almost all calculators have built-in square root functions, for instance; you punch in the number 9, press the square-root symbol, and the calculator displays the number 3. BASIC has a large collection of such numeric functions. You'll learn about two of them in this session, one for generating random numbers (RND) and one for getting rid of the fractional part of numbers (INT).

## The RND Function

Randomness is the basis of many of life's twists and turns. A winning roll of the dice, a meeting that leads to a beautiful romance, a grade point average based on multiple-choice answers all have their roots in random chance. Many scientific experiments involve random numbers, as does the military draft and, to a large extent, Internal Revenue Service audits. BASIC lets you generate your own random numbers by using one of its numeric functions, **RND**.

Most BASIC functions work the same way. You type the function's name and follow it with its **argument** (that which the function is to operate upon) enclosed in parentheses. The RND function returns a **real number** (a number with a decimal part) between zero and the argument you give it.



### Do This

Type in a sample program to demonstrate RND.

1. Get a new listing window by typing ⌘ N.
2. Type this in:

```
FOR Chance = 1 TO 25  
  PRINT RND(5)  
NEXT Chance
```

3. Execute the program. Figure 7-1 shows the listing and the sort of output you'll get.
4. Scroll through the output window, noting that all of the numbers are greater than zero and less than 5.
6. Get the listing window back and change RND's argument to 10.
7. Execute the program again, noting this time that the numbers are all less than 10.

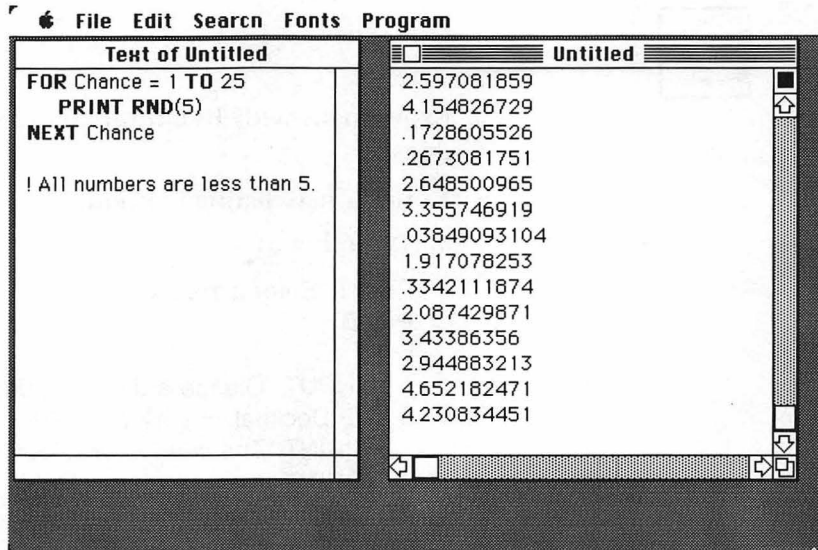


Figure 7-1 RND chance

Quite often random numbers are more useful when they're **integers** (numbers with no fractional or decimal parts). The next section tells you about a function that converts real numbers (that is, decimal numbers) to integers.

### The INT Function

The **INT** function converts decimal numbers into integers by cutting off any decimal part and returning the next lower whole number. Its form is just like RND's: you follow the function name with an argument (in this case, the number you want made into an integer) enclosed in parentheses. The program in the next "Do This" box uses INT to convert any decimal number you enter into an integer.





### Do This

Experiment with INT until you have a handle on what it does.

1. Get a new listing window.
2. Type this in:

```
PRINT "Enter a zero to end."  
PRINT  
DO  
    INPUT "Gimme a decimal number: "; Decimal  
    IF Decimal = 0 THEN EXIT  
    PRINT "The integer for "; Decimal; " is "; INT(Decimal)  
    PRINT  
LOOP
```

3. Run the program.



### For Your Information

Note that the argument for INT (and all other numeric functions in BASIC) can be a variable as well as a constant.

The program prompts you for a decimal number. When you type it in, BASIC assigns the number to the INPUT variable called Decimal. That variable is used as the argument for the INT function, which converts Decimal's value to a whole number or integer. The business part of the program is in the DO\LOOP construct, which is set up to repeat forever if necessary. If you enter a zero, BASIC leaves the loop. Having no more statements to execute, the program then stops.

### “Why Am I Doing This?”

All of this is indeed leading somewhere. By the time you're done with this book, you'll have used this information to write (among other programs) two popular games: the dice game *Craps* and the justifiably famous *Great American Sheep Race*. Play things right and in no time at all you'll get a vacation home in Cannes.

First, as a self-test to see how you're doing, you'll combine the integer and random number functions into the same program, based on specifications I'll give you. Later on, you'll apply what you learned in that process to a graphics program. Finally, you'll work on the *Craps* program (you'll write the *Sheep Race* program in Session 12).

### RANDOMIZE for More Genuine Randomness

RND doesn't produce really random numbers. Every time you run the following program, for example, you'll get exactly the same results:

```
FOR PseudoRandom = 1 TO 10
  PRINT RND(10)
NEXT PseudoRandom
```

Run it twice to see what happens; Figure 7-2 shows the results I got.

Untitled	Untitled
7.826369256E-5	7.826369256E-5
1.315377881	1.315377881
7.556053218	7.556053218
4.586501317	4.586501317
5.327672372	5.327672372
2.189591862	2.189591862
.4704461619	.4704461619
6.788647166	6.788647166
6.792964055	6.792964055
9.346928955	9.346928955

**Figure 7-2** First and second run of RND function

A lot of scientific experiments need repeating series of random numbers, but most games and graphics programs don't. If you want your programs to produce different numbers each time you use RND, start your programs with the keyword **RANDOMIZE**, like this:

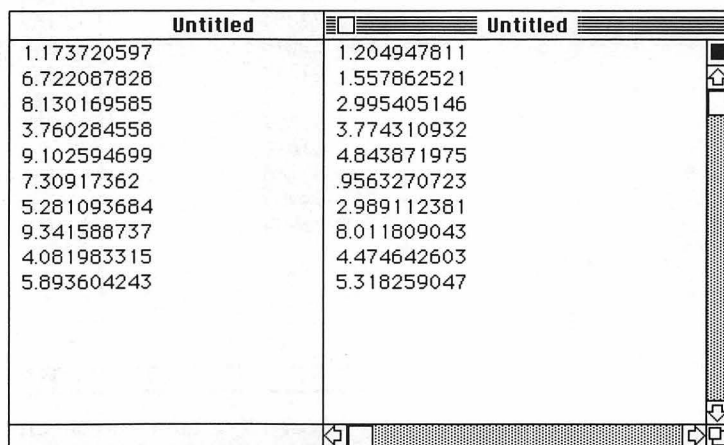
```
RANDOMIZE
FOR PseudoRandom = 1 TO 10
    PRINT RND(10)
NEXT PseudoRandom
```

Figure 7-3 shows the output of two runs of this program.

### Integer Randomness

Write a program that produces 50 random integers, all of which are in the range 1 through 100 inclusive. There doesn't have to be either a 1 or 100 in the list; it just must be possible for the program to produce them. The catch is that you can't use any IF...THEN statements.

That presents a problem or two. The random function sometimes produces numbers between 0 and 1. If you feed a decimal number less than 1 to the INT function, INT returns 0; but zeros aren't allowed in this assignment—the number 1 is the lowest allowable. I give my solution in the next “Do This” box, but don't look there until you've tried on your own. You have a 15-minute time limit.



Untitled	Untitled
1.173720597	1.204947811
6.722087828	1.557862521
8.130169585	2.995405146
3.760284558	3.774310932
9.102594699	4.843871975
7.30917362	.9563270723
5.281093684	2.989112381
9.341588737	8.011809043
4.081983315	4.474642603
5.893604243	5.318259047

**Figure 7-3** First and second run of RND function with RANDOMIZE

**Stop!** If you read the solution before attempting to solve the problem yourself, a power surge will burst through your computer and reduce it to a useless pile of silicon chips.



### Do This

Write a program that produces 50 random whole numbers, all of which are between 1 and 100 inclusive.

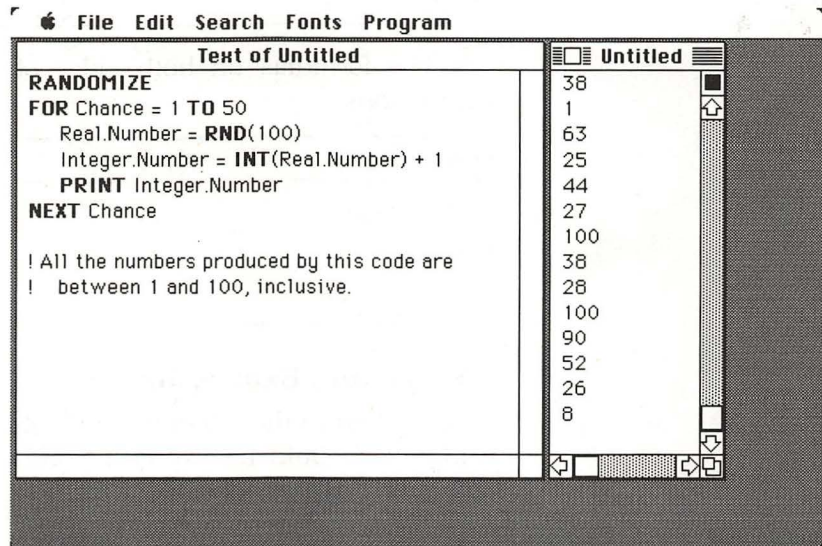
```

RANDOMIZE ! just to keep things from getting boring
FOR Chance = 1 TO 50
  Real.Number = RND(100)
  Integer.Number = INT(Real.Number) + 1
  PRINT Integer.Number
NEXT Chance

```

Figure 7-4 shows the kind of result you'll get.

Here's how the program works. The FOR\NEXT loop goes from 1 to 50, giving me the 50 rounds I need. I get a random decimal number less than 100 and assign it to the variable



**Figure 7-4** Fifty random whole numbers



Real.Number. Then I convert the decimal number to a whole number, and (here comes the sneaky part) I also add 1 to it. That way, if the number is 0, it becomes 1, and if it's 99, it becomes 100—so all numbers will be between 1 and 100, as ordered.

## Expressions

In almost every case in BASIC where you can use a numeric constant (including the argument of a numeric function) you can use either a variable or an **expression** instead. An expression is any group of constants and/or variables coupled with operators and/or functions that is meant to be taken as a single value. For example:

INT(Real.Number) + 1

This.Value \* 37

My.Pay - State.Tax - Federal.Tax - Thumb.Tax

When you set up an assignment statement, everything to the right of the assignment operator = is an expression:

Variable Name		Expression
Real.Number	=	RND(100)
Integer.Number	=	INT(Real.Number) + 1
Federal.Tax	=	My.Pay * .58

The formulas on both sides of relational operators are expressions:

Expression	Operator	Expression
This + That	<	The.Other.Thing - 12
INT(Number)	> =	Base.Count
Something / 87	=	Count - Total + Range

## Compacting Expressions

The solution to the “50 random integers” problem that you just worked on could be expressed in a more compact form by combining expressions, like this:

Expanded

```
Real.Number = RND(100)
Integer.Number = INT(Real.Number) + 1
PRINT Integer.Number
```

Contracted

```
PRINT INT(RND(100)) + 1
```

The statement on the right produces the same result as the three statements on the left. It has the advantages of being quicker to type and taking up less computer memory. It has the disadvantages of being more complex, harder to read and to follow, and more prone to typing mistakes—leave out one parenthesis and it's Error Message City. I prefer the form on the left for teaching; I tend to use the form on the right for programming.

**Precedence and Nested Expressions**

The compacted expression you just saw demonstrates how one expression can be nested within another one. It's easy for BASIC to figure out what you want it to do with the two expressions because of BASIC's rules of **precedence**. Precedence has to do with the order of priorities in a system; in BASIC, precedence determines what expressions are calculated first in a given formula. Table 7-1 shows a complete list of the order of precedence.

**Table 7-1** Rules of Precedence

Operator	Comment
( )	In case of nesting, innermost item is evaluated first
+ -	Unary operators (+ 6, - 12)
* /	Standard multiplication and division
+ -	Standard addition and subtraction
= < >	Relational operators (not the assignment operator =)

Operators high on the list have priority over those lower on the list; that is, they are carried out first. Operators on the same line in this list are carried out in order from left to right as they appear in an expression. Here's an example:

```
10 * 36 - 12 / 2 + 6 - 100 / 50
```

The formula looks like this to BASIC, since BASIC does multiplication and division before addition and subtraction:

$$(10 * 36) - (12 / 2) + 6 - (100 / 50)$$

Here's how BASIC evaluates the results:

$$360 - 6 + 6 - 2 = 358$$

If you change the formula with parentheses of your own, BASIC evaluates it differently, solving your parenthetical expressions first:

$$\begin{array}{rcll} 10 * (36 - 12) / 2 + (6 - 100) / 50 & & & \\ 10 * 24 / 2 + -94 / 50 & & & \\ 240 / 2 + -1.88 & & & \\ 120 + -1.88 & = & 118.12 & \end{array}$$

Finally, if you have nested parenthetical expressions, BASIC solves them from the inside out:

$$\begin{array}{rcll} 10 * (36 - (12 / 2 + 6) - 100) / 50 & & & \\ 10 * (36 - (6 + 6) - 100) / 50 & & & \\ 10 * (36 - 12 - 100) / 50 & & & \\ 10 * -76 / 50 & = & -15.2 & \end{array}$$

Don't take my word for all this, though; I've been known to lie. Before you go on, experiment with precedence by writing a few one-line test programs. Just precede any expression you devise with the keyword PRINT; your Macintosh and BASIC will do the rest.



## Pop Quiz

### Question 1

Using INT and RND, write a program that produces prices for 20 different computer books made to sell somewhere between \$5 (ah—the old days!) and \$25.99 inclusive. All you need to show are the 20 prices. Of course, prices include dollar signs and decimal points. And there's no such thing as a book that costs \$12.657888343—so allow a maximum of two numbers after the decimal point, please! Some of your prices might not have two numbers after the decimal point (like \$9.6) or may be in whole dollars (\$7); that's OK for now. Watch yourself—this one is tricky!

## Graphic Randomness

One of the versions of the *Little Boxes* program from Session 6 looked like this:

```
DO
  FRAME RECT MOUSEH - 20, MOUSEV - 20; MOUSEH, MOUSEV
  IF MOUSEV < 0 THEN CLEARWINDOW
  IF MOUSEB = 1 THEN EXIT
LOOP
PRINT "That's it!"
```

Time to change it to add some randomness. Do it on your own, following the suggestions in the next box.



### Do This

Rewrite the *Little Boxes* program so that it does essentially the same thing—draws little boxes on the screen, each one 20 dots on a side—but with certain modifications:

- use variables wherever you can
- let random numbers decide where the boxes will appear

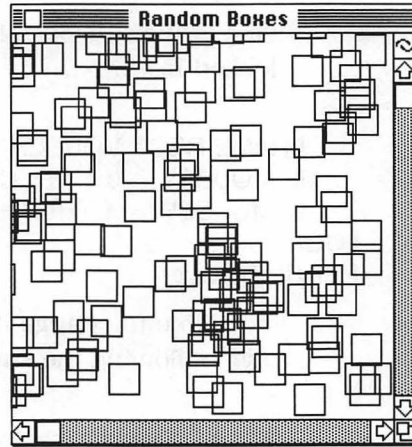
You don't need to convert the real numbers produced by RND to integers when you use them for graphics (although you'd think that you'd have to—it's tough to plot a point at row 12.345555443, column 127.34532). BASIC does it for you.

If you're stuck after about 15 minutes, go on to read my solution.

### A Possible Solution

Here's my solution to the "Random Boxes" problem. If yours works and doesn't look anything like this one, then your solution is every bit as good as mine (and quite possibly a lot more creative). Figure 7-5 shows my program's output. Here's the program listing:





**Figure 7-5** Random boxes

Extreme.Col = 240	! Last column in standard output window
Extreme.Row = 240	! Bottom row in standard output window
Box.Size = 20	! Number of dots on a side
RANDOMIZE	
DO	
Low.Col = RND(Extreme.Col)	! Choose this box's right side location
Low.Row = RND(Extreme.Row)	! Choose this box's bottom location
FRAME RECT Low.Col - Box.Size; Low.Row - Box.Size; Low.Col, Low.Row	
IF MOUSEV < 0 THEN CLEARWINDOW	! Move mouse over the top to erase
IF MOUSEB = 1 THEN EXIT	! Press mouse button to end
LOOP	

Note the three variables set outside the DO\LOOP construct. They're outside the loop because they don't depend upon anything inside the loop to modify their values; once they're set, they don't need to be reset. As such, they're called constants because their values don't change.

### Random Bugs

Of all the bugs that crawl into a random number program like this, the most common ones deal with when to get a new random number and when not to. The next exercise will show several of them.



## Do This

If the effect of any of the following bugs doesn't become intuitively obvious to you as I discuss it, change the *Random Boxes* program so that it has that bug and then run the program so you can see the results. Finally, restore the program so you can try the next bug. *Please don't skip this!* It'll pay off in the long run.

1. You have to make sure that you get a new random number before you draw each box. If you set the random number outside the loop, BASIC will draw the same box over and over again (if it draws one at all).

```
....
Box.Size = 20
RANDOMIZE
Low.Col = RND(Extreme.Col)
Low.Row = RND(Extreme.Row)
DO
    FRAME RECT ...
...

```

2. You have to make sure that your random numbers are in the right range; otherwise you'll be drawing lots of boxes out of the window. For instance, try this coordinate set:

```
Extreme.Col = 5000
Extreme.Row = 5000
```

3. Once RND has established the row and column for the box position, you have to use the same values throughout the line. Otherwise, really strange things happen. (The following code should be typed all on one line in the computer; it appears as two lines because of the limitations of the printed page.)

```
FRAME RECT RND(Extreme.Col) - Box.Size, RND(Extreme.Row) -
Box.Size; RND(Extreme.Col), RND(Extreme.Row)
```

Remember that RND produces a new random number each time you call for one. If you want to use the same random number in different expressions, you must assign the random number to a variable (as in the preceding example).

### A Few Final Modifications

Before you leave the *Random Boxes* program, here are a few more changes you can make in it just for fun:

- Change the variable Box.Size from 20 to a random number—say RND(50)—and move it just inside the loop.
- Change FRAME RECT to INVERT OVAL.

Here's what the whole thing looks like with the changes in place:

```
Extreme.Col = 240
Extreme.Row = 240
RANDOMIZE
DO
  Box.Size = RND(50)
  Low.Col = RND(Extreme.Col)
  Low.Row = RND(Extreme.Row)
  INVERT OVAL Low.Col - Box.Size, Low.Row - Box.Size; Low.Col, Low.Row
  IF MOUSEV < 0 THEN CLEARWINDOW
  IF MOUSEB = 1 THEN EXIT
LOOP
```

Your output should look something like that in Figure 7-6. Change these elements either one at a time or in whatever combination you want. The important thing, of course, is to experiment.

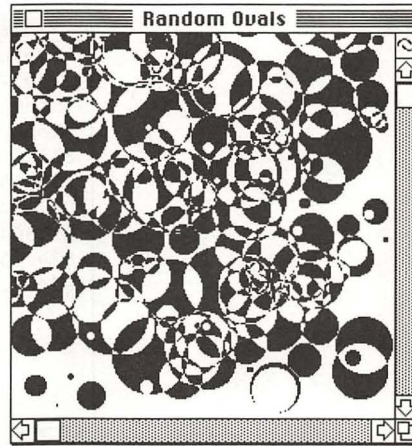


Figure 7-6 Random INVERTed OVALs



### For Your Information

**Teeny RND** If you want all the numbers that RND produces to be between 0 and 1, don't give RND an argument:

```
FOR Teeny = 1 TO 10  
  PRINT RND  
NEXT Teeny
```

You'll get something like Figure 7-7.

## Subroutines—For Programs That Are Easier to Read and Easier to Fix

BASIC has a branching instruction—a keyword that redirects the flow of control to some other part of the program—called **GOSUB**. This keyword is always followed by a **label**, a string of text characters beginning with a number or letter and ending with a colon (:). The GOSUB statement is made up of the keyword GOSUB and its label.



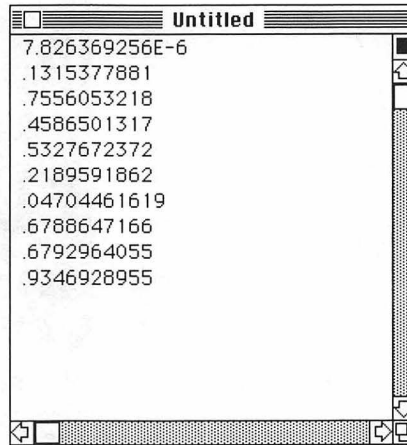


Figure 7-7 Teeny RND

When BASIC comes across a GOSUB instruction, it reads the label following GOSUB and then searches through the program until it comes across a section of code beginning with the same label. It executes whatever code it finds there until it comes to the keyword **RETURN**. This keyword sends BASIC back to the end of the GOSUB statement, and BASIC continues its execution from there. The block of code between the label and RETURN is called a **subroutine**.

New status icon:  
Question mark shows  
program is waiting for  
input from user.

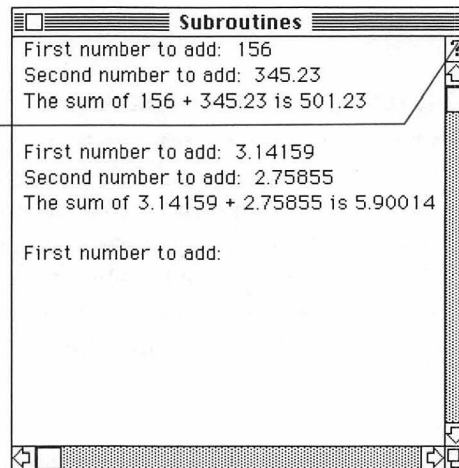


Figure 7-8 Branching with GOSUB



### Do This

Type in the following code exactly as it appears. (You don't have to type in the comments.) Then run the program.

```
FOR Round = 1 TO 5
  INPUT "First number to add: "; First.Num
  INPUT "Second number to add: "; Second.Num
  GOSUB Add.Em.Up:    ! Label ends with a colon
  PRINT               ! Blank line displayed after
                     ! the RETURN happens

NEXT Round
END PROGRAM           ! I'll come back to this.

Add.Em.Up:           ! Here's the subroutine code block.
  Total = First.Num + Second.Num
  PRINT "The sum of "; First.Num; " + "; Second.Num; " is "; Total
RETURN               ! This sends control back
```

The results of this program look like Figure 7-8.

Here's how the program works: BASIC enters the loop and gets two numbers from the computer operator. Then it branches to the subroutine called Add.Em.Up, performs the addition, and shows the result. BASIC now comes to RETURN and so goes back up to execute the next statement (which displays a blank line), and then to repeat the loop.

Here's another version of the same program, using three subroutines.



### Do This

Replace the program in the listing window with the following code, then run the program.

```
FOR Round = 1 TO 5
  GOSUB Get.Two.Numbers:
  GOSUB Add.Em.Up:
  GOSUB Show.Results:
  PRINT
NEXT Round
END PROGRAM

Get.Two.Numbers:
  INPUT "First number to add: "; First.Num
  INPUT "Second number to add: "; Second.Num
RETURN

Add.Em.Up:
  Total = First.Num + Second.Num
RETURN

Show.Results:
  PRINT "The sum of "; First.Num ; " + "; Second.Num; " is "; Total
RETURN
```

Of all the programs you've typed into your computer so far, this program is probably the easiest to follow. To know what the program does, all you need do is read the contents of the FOR\NEXT block. It says, "Get two numbers from the computer operator. Add the two numbers together. Show the results of the addition. Put in a blank line for neatness. Repeat this procedure a total of five times."

The labels tell you what each section of the code does. Now, if you run into a problem with the code—if there's a bug in an INPUT line, if the numbers don't add up right, if the display is messed up or whatever—you (or any other programmer using your code) know exactly where to look to solve the problem.



Subroutines really show their worth when you have a long block of code that is used repeatedly in the same program. Rather than retyping the code every time you need it, you just type it in once as a subroutine, giving it a label name like `Code.Block:`. The next time you need it, just type `GOSUB Code.Block:` and BASIC will take care of the rest.

### About That **END PROGRAM** . . .

The examples you just used contained a new keyword, **END PROGRAM**. **END PROGRAM** simply tells BASIC to stop executing code; as you might guess, it means “This is the end of the program.” Using **END PROGRAM** is optional in many cases; you haven’t needed it at all up to now. Ordinarily, a program stops automatically when BASIC runs out of code to execute.

You use **END PROGRAM** to protect BASIC from accidentally “falling into” a block of code. You read earlier that `GOSUB` sends program flow to a part of the program that starts with a particular label. But you read earlier still that flow proceeds sequentially through a program unless it’s redirected by some keyword. If you take out the **END PROGRAM** in the previous example, BASIC finishes the rounds and continues merrily right into the first subroutine, eventually executing the `RETURN` statement. But it doesn’t know where to return to, because it wasn’t sent to the subroutine via a `GOSUB` statement; it simply stumbled into it!



### Do This

Remove the **END PROGRAM** statement from the preceding addition program; then execute the program again. Figure 7-9 shows what you’ll get.

It’s always a good idea to put all your subroutines at the end of your program and precede them with the keyword phrase **END PROGRAM**.



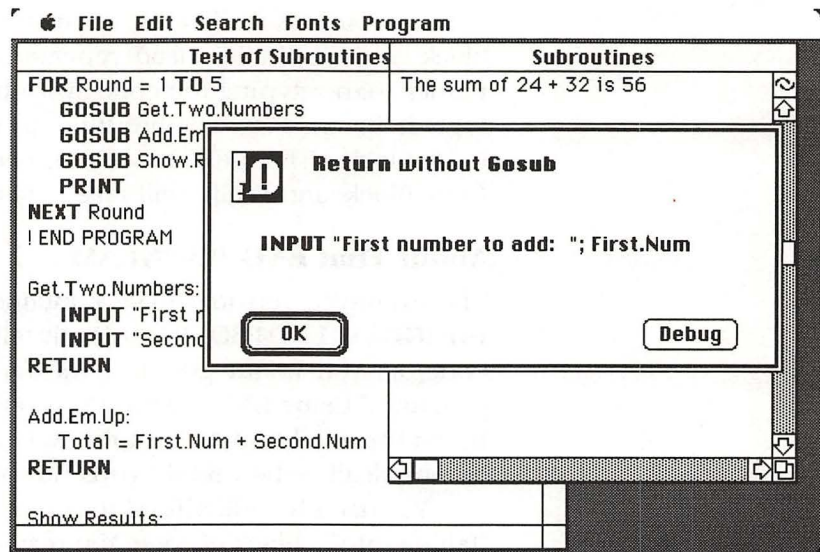


Figure 7-9 "RETURN without GOSUB" error message



## For Your Information

**About Those Labels** Actually, you don't need a colon after the label in the **referencing line** (the line that has the GOSUB keyword); you just need it after the label at the beginning of the subroutine which marks GOSUB's destination. I always use colons after labels so I don't confuse them with variable names, but you can do as you like.

Play with subroutines for a while, until you're sure you understand how they operate. Then go on to the next section to find out how much you've learned so far.

## The Dice Game

---

You now have a good enough command of BASIC to write a fairly complex program. Here's the opportunity to put your knowledge to the test and at the same time to plan and write something really useful—a computerized version of the ancient dice game, craps.

To write the game program, you'll need to use most of what you've learned so far: GOSUB, RETURN, END PROGRAM, DO\LOOP, IF...THEN, EXIT, PRINT, CLEARWINDOW, RANDOMIZE, assignment and relational operators, strings, and the two numeric functions RND and INT. Later you might even want to add graphics.

### Game Rules

Here's how the game works. You begin with a pair of dice, six sides to each die. Each side has from one to six dots or *pips* on it. You roll the dice and add up the pips; the total can be anywhere from 2 through 12.

Different rules apply for the first roll than for all subsequent rolls. If your pips total 7 or 11 on the first roll, you win; if they total 2, 3, or 12, you lose. If the total is any other number—4, 5, 6, 8, 9, or 10—that number is your *point*. You continue to roll the dice until either the total of pips matches your point, in which case you win, or until you roll a 7, in which case you lose. If you win, you get to roll again; if you lose, you must pass the dice to the next player.

Members of the lower classes tend to wager on the game's outcome.

### Planning the Program

You can outline the description of the dice and the rules of the game in such a way that you'll end up with an excellent program plan. I do it here for you; all you have to do is write the code (simple, huh?). Just keep in mind that all computers, including the Macintosh, are incredibly stupid; you have to tell them nearly everything.

1. A roll of a die can produce any of 6 possible integers, in the range 1 through 6.
2. You roll two dice at a time.
3. The sum of the pips on the dice for each roll is important; you have to keep track of it.
4. On the first roll, the sums 7 and 11 win, while 2, 3, and 12 lose.
5. If you don't win or lose on the first roll, the sum of the pips on the first roll is your "point."
6. If the sum of the pips on the next roll is the same as your point, you win; if the sum is 7, you lose.
7. If you don't either win or lose on this roll, you repeat step 6.
8. If you win, you get to play again; if you lose, you pass the dice.

Two added niceties that you ought to consider are not in the outline. First, if you aren't the only person who will use your program, you might want to add instructions both about the game of craps and about how to make your program work. Second, you might want to include a way to end the game (no matter how much fun a game is, people eventually have to do things like eat and sleep). If you're really depraved, you can add betting instructions and features to the program, including facilities for keeping totals for the players.

### **Some Suggestions**

To keep things simple, I suggest that you set up the game for just two players; later, when you learn about something called arrays, it'll be easy to change the program to work for any number of people. I also suggest that you use subroutines wherever you can. Remember to use meaningful labels and variable names.

As is my wont, I have given you a challenging assignment (I don't want to bore you with trivial problems). Therefore, my final suggestion is that you give yourself plenty of time to plan, write, and debug the program. Nobody's keeping score except you. Since you'll need most of your knowledge to write this code, it'll be a great self-test for you.

My solution is included in the Summary section. As usual, there are a huge number of ways to write this program; my solution is just one of many. Good luck!

## Summary

---

### New Terms

---

**Algorithm** a step-by-step procedure for solving a problem.

**Argument** the expression (in parentheses) that a given function is to operate upon.

**Branching instruction** an instruction to BASIC to transfer flow of control to another part of the program.

**Expression** any group of values to be taken as one value. Expressions can include constants, variables, functions, and operators.

**Integer** number with no fractional or decimal part; a whole number.

**Label** either a number or a string of text characters beginning with a letter and ending with a colon, identifying the beginning of a subroutine. Also used in a GOSUB statement as a subroutine's name.

**Numeric function** function designed to manipulate a number or numeric expression. A numeric function usually takes one argument (enclosed in parentheses) and returns a single result.

**Precedence** the order in which BASIC performs calculations in a given expression or formula.

**Real number** number with a decimal part.

**Referencing line** a line of code that contains a branching instruction (such as GOSUB). It's called a referencing line because it *refers to* a label that begins a subroutine in some other part of the program.

**Subroutine** program block beginning with a label and ending with the keyword RETURN. BASIC executes a subroutine when it comes across the keyword GOSUB immediately followed by the subroutine's label.

**Unary operator** an operator that applies to a single value. For example, the minus sign in  $-7$  is said to be *unary* because it defines 7 as a minus number, as opposed to expressing an operation to be performed on two values (like  $x - 7$ ).

### Programming Statements

---

**END PROGRAM** immediately halt execution of program. You use it to keep program control from inadvertently falling into a subroutine code block. The keyword PROGRAM is optional.

**GOSUB label:** redirect program control to a subroutine whose code block begins with the single word *label:* and ends with the keyword RETURN.



**INT** a numeric function returning the next lower whole number value of its argument. (For example, 87.86 returns 87;  $-87.86$  returns  $-88$ .)

**RANDOMIZE** scramble the random number generator so that each time you run a program with RND in it, you get different random numbers.

**RETURN** restore program control to the statement immediately following the one containing the keyword GOSUB that redirected program control to this subroutine.

**RND** a numeric function that returns a random real number between zero and its argument.

---

## Pop Quiz Answer

---

### Question 1

Here's my solution. The problem was a tough one; if you didn't get it, don't worry about it. Look at the way I did it and play with it until you understand it. Let the computer help you!

```
RANDOMIZE
FOR Price = 1 TO 20
    Random.Number = RND(21)
    Real.Price = INT(Random.Number * 100) / 100
    Adjusted.Real.Price = Real.Price + 5
    PRINT "$"; Adjusted.Real.Price
NEXT PRICE
```

Alternatively; use this:

```
RANDOMIZE
FOR Price = 1 TO 20
    PRINT "$"; INT(RND(21) * 100) / 100 + 5
NEXT Price
```

The second listing is a highly condensed version of the first listing.

If you're having trouble understanding my solution, look at Figure 7-10; it shows the listing window and an output window of a program that demonstrates how the **algorithm** (step-by-step problem-solving method) for my solution works. It produces one price; put Figure 7-10's listing in a FOR\NEXT loop to see it produce all 20 prices.

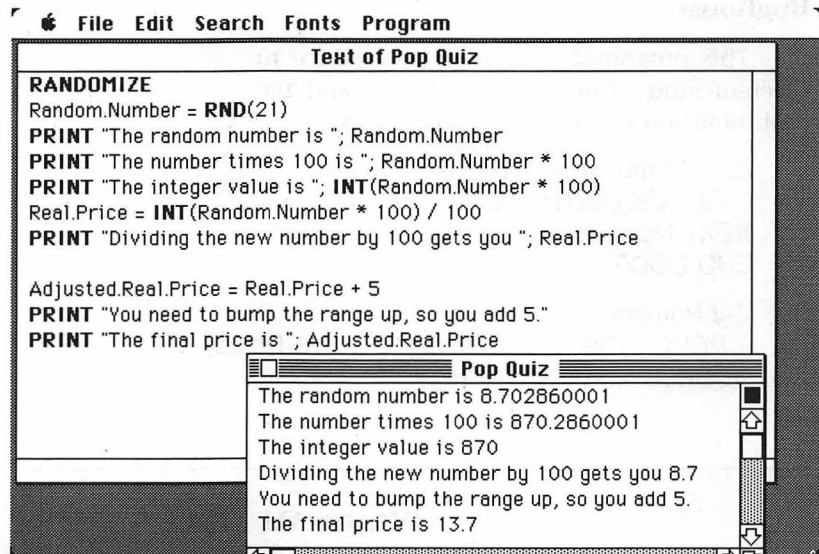


Figure 7-10 Answer to Pop Quiz

## Commands, Menu Items, Keywords You Know So Far

### Commands and Menu Items

Backspace key	Paste
Clear	Run
Copy	Save
Cut	Select All
Halt	Undo
New	⌘ key
Open	

### Programming Characters

+	-	/	*	=
>	<	;		
!	\$			

### Programming Statements

BTWAIT	MOUSEB
CLEARWINDOW	MOUSEH
DO\LOOP	MOUSEV
ELSE	OVAL
END	PAINT
EXIT	PLOT
FOR...TO...STEP\NEXT	PRINT
FRAME	RANDOMIZE
GOSUB	RECT
IF...THEN	RETURN
INPUT	RND
INT	
INVERT	

---

**Bughouse**

This program is supposed to produce 10 different random numbers between 1 and 100 each time you run it. It's got seven bugs.

```
FOR Numbers = 1 TO 10
  GOSUB Get.Numbers
NEXT Number
END CODE

Get.Number
  PRINT "This random number is "; RND(100)
GOBACK
```

---

**PROGRAM : Dice Game**  
**by Throckmorton Scribblemonger, Ph.D. (cand.)**  
**10:37:25 AM 6/11/84**  
**V 1.1**

```
DO
  GOSUB Initialize:
  GOSUB Instruction.Request:
  GOSUB Rollem:
  GOSUB Check.First.Roll:
  Point = Roll
  GOSUB Formal.Game:
  GOSUB Another.Game:
  IF Done = 1 THEN EXIT
LOOP

PRINT "Thanks for playing."
END PROGRAM

Another.Game:
  INPUT "Enter Return to go on; anything else to stop :"; Question$
  IF Question$ <> "" THEN Done = 1
  CLEARWINDOW
RETURN

Button.Push:
  PRINT "Push the button to roll the dice."
  BTNWAIT
  PRINT
RETURN
```

*(Listing continued on next page.)*

Check.First.Roll:

```
IF Roll = 7 THEN GOSUB Winner:
IF Roll = 11 THEN GOSUB Winner:
IF Roll = 2 THEN GOSUB Loser:
IF Roll = 3 THEN GOSUB Loser:
IF Roll = 12 THEN GOSUB Loser:
```

RETURN

Formal.Game:

```
DO
  IF Endit = 1 THEN EXIT
  GOSUB Button.Push:
  PRINT "Your point is "; Point
  GOSUB Rollem:
  GOSUB Win.Or.Lose:
```

LOOP

RETURN

Initialize:

```
RANDOMIZE
Point = 0
Roll = 0
Endit = 0
```

RETURN

Instruction.Request:

```
INPUT "Want instructions? (y/n--Return means 'n'): "; Request$
IF Request$ = "y" THEN GOSUB Instructions:
CLEARWINDOW
```

RETURN

Instructions:

```
PRINT "This is the dice game CRAPS. Push the "
PRINT "button on the mouse to roll the dice."
PRINT "If your first roll is a 7 or 11, you win."
PRINT "If your first roll is a 2, 3 or 12, you lose."
PRINT
PRINT "The total of the dice is your point. Roll"
PRINT "your point and you win; roll a 7 and you lose."
GOSUB Button.Push:
CLEARWINDOW
```

RETURN

*(Listing continued on next page.)*



Loser:

PRINT "Sorry, friend. Pass the dice."

Endit = 1

RETURN

Roller:

Die.1 = INT(RND(6)) + 1

Die.2 = INT(RND(6)) + 1

Roll = Die.1 + Die.2

PRINT "You rolled "; Die.1; " and "; Die.2; "--"; Roll

PRINT

RETURN

Winner:

PRINT "Yea! A winner! You keep the dice!"

Endit = 1

Winner = 1

RETURN

Win.Or.Lose:

IF Roll = Point THEN GOSUB Winner:

IF Roll = 7 THEN GOSUB Loser:

RETURN

---

# SESSION



## Strings, Mostly

---

**Y**ou've been using strings for the last few sessions, but up to now you haven't been able to do much with them. In this session you'll learn a lot more about strings. You'll work with string functions, numeric functions related to strings, an international computer coding system known as ASCII, the Alarm Clock and Key Caps desk accessories, and general bits and pieces that make working with Macintosh BASIC fun and exciting. The session probably won't equal the excitement that skydiving brings to your life, but you'll learn a lot.

### **The LEN Function—Counting Characters**

The LEN function returns the number of characters in a string. To remind you, a string is any sequence of text characters; spaces and punctuation marks are often parts of strings.



### Do This

Set up a new listing window for experimenting with string functions.

1. Type `⌘ N` to get a new listing window.
2. Since some of the program lines will be long, make the listing window larger by dragging the size box to the right.
3. Type in this short program:

```
Test$ = "What time is lunch?"  
Length = LEN(Test$)  
PRINT "Here's the string: "; Test$  
PRINT "There are "; Length; " characters in the string."
```

4. Predict what number BASIC will give you for the variable `Length`.
5. Execute the program.

## Substring Functions

Sometimes it's useful to manipulate parts of strings, or **substrings**. BASIC has several functions to deal with substrings. **LEFT\$** lets you handle any number of characters from the beginning of a string (the left side), and **RIGHT\$** deals with the end of a string (the right side). A third function, **MID\$**, manipulates the middle characters.

Unlike `INT`, `RND`, and `LEN`, all of which take just one argument (that is, one item enclosed in parentheses), these new string functions take at least two arguments separated by commas; `MID$` sometimes takes three.



## Do This

Modify the program in memory to experiment with `LEFT$` and `RIGHT$`.

1. Add the following lines to your program:

```
PRINT "Here are the first four characters: "; LEFT$(Test$, 4)
PRINT "And here are the last four: "; RIGHT$(Test$, 4)
```

2. Before running the program, predict what will happen.

3. Run the program.

Figure 8-1 shows what the whole program looks like and what it produces.

The first argument, always a string constant or string variable, tells the function what string you're dealing with; the second argument, which must be a numeric constant, numeric variable, or numeric expression, says how many characters you want.

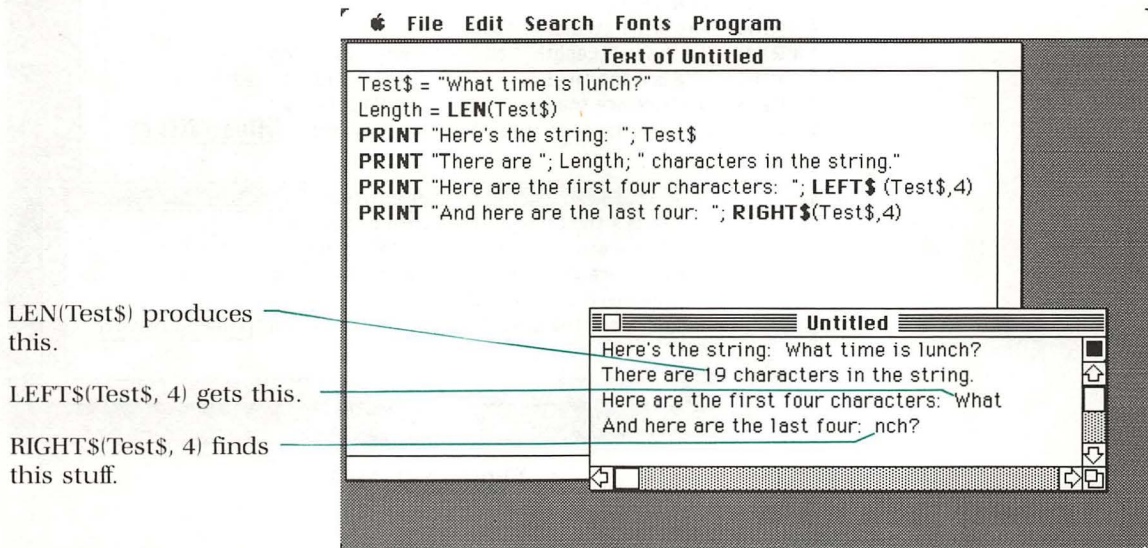


Figure 8-1 `LEN`, `LEFT$`, `RIGHT$`



MID\$ works somewhat differently; its second argument tells it from which position in the string, counting from the left, to start returning characters. If you don't give a third argument, MID\$ returns all the remaining characters.



### Do This

Add this line to the program and run it:

```
PRINT "Here are the ones from the 6th to the end: "; MID$(Test$, 6)
```

Figure 8-2 shows the results.

If you give MID\$ a third argument (which can be any number, numeric variable, or numeric expression), you'll get all the characters beginning at the position specified by the second argument through the character specified by the third argument.

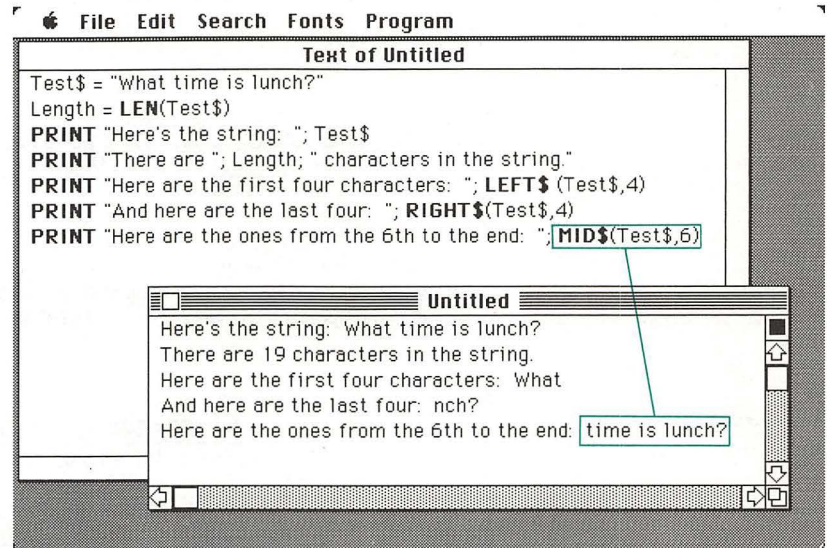


Figure 8-2 MID\$





### Do This

Add this final line to your program and run it:

```
PRINT "Here are characters number 6 though 9: "; MID$(Test$, 6, 4)
```

These substring functions have practical uses, especially for searching out special characters or groups of characters. For instance, I can't stand computer programs that ask me to type in my last name and then my first name; instead, the following example asks for the operator's full name and then extracts just the first name from it.

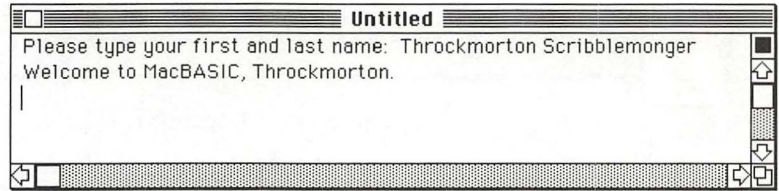


### Do This

Type this program into a clean listing window, determine what purpose each line has (playing computer will help a lot here), and then execute the code. The quote marks in the fifth program line have a space character between them:

```
Search = 0
INPUT "Please type your first and last name: "; Name$
DO
    Search = Search + 1
    IF MID$(Name$, Search, 1) = " " THEN EXIT
LOOP
First.Name$ = LEFT$(Name$, Search - 1)
PRINT "Welcome to MacBASIC, "; First.Name$; "."
```

Figure 8-3 shows what you should get.



**Figure 8-3** Result of character examination using MID\$

Here's how the program works. BASIC looks at each character in Name\$ until it finds a space character:

```
DO
    Search = Search + 1
    IF MID$(Name$, Search, 1) = " " THEN EXIT
LOOP
```

Since Search increments by 1 each time through the loop, its value is the position in Name\$ of the particular character BASIC is looking at; thus, when BASIC exits the loop after having found the space character, Search holds the space character's position. In this case, the space character is in position 13. To find just the first name in Name\$, all BASIC has to do is take out all the characters up to (but not including) the space character:

```
First.Name$ = LEFT$(Name$, Search - 1)
```



## Pop Quiz

### Question 1

Using the RIGHT\$ function, what single line of code can you add to extract and display just the last name out of Name\$ from the above example?

**And Just for the Exercise . . .**

Type in and run the following program; then experiment with the three substring functions by changing the code in this example until you really understand how substring functions work.

**Do This**

Type this program into a clean listing window, predict what will happen, and run it.

```
Test$ = "This Macintosh is a fabulous machine!"
FOR String = 1 TO LEN(Test$)
    PRINT LEFT$(Test$, String)
NEXT String

PRINT
PRINT MID$(Test$, 21, 8)
PRINT

FOR String = 1 TO LEN(Test$)
    PRINT RIGHT$(Test$, String)
NEXT String
```

Figure 8-4 shows some of the results.



**Figure 8-4** Fabulous strings



## Concatenation: Joining Strings Together

**Concatenation** is like substrings in reverse; substrings let you tear a single string apart, while concatenation lets you pull several strings together. You learned in an earlier session that you can perform certain arithmetic operations on numeric variables—that is, you can add, subtract, multiply and divide them. While you can't add strings together (Zen question of the day: What's the sum of *dog* and *pony*?), you can join them through concatenation. The concatenation operator is the ampersand character, &. You can use it with string literals, string variables, or a combination of the two.

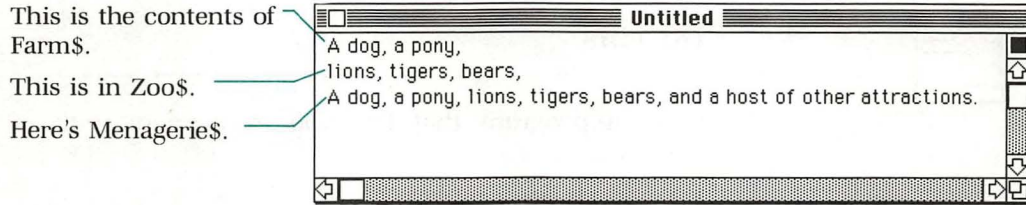


### Do This:

Clear out a listing window, then type in and execute this program:

```
Animal.1$ = "A dog, "  
Animal.2$ = "a pony, "  
Farm$ = Animal.1$ & Animal.2$  
PRINT Farm$  
  
Zoo$ = "lions, " & "tigers, " & "bears, " ! Oh, My!  
PRINT Zoo$  
  
Menagerie$ = Farm$ & Zoo$ & "and a host of other attractions."  
PRINT Menagerie$
```

Figure 8-5 shows the output window for this program.



**Figure 8-5** String concatenation

## String Comparisons

You've already used the relational operators ( $<$ ,  $>$ ,  $=$ ) to make comparisons between numbers. You can use the same operators in the same way to compare strings:

```
This$ > That$  
Myth$ < "Put that globe down, Atlas!"  
"Not with a bang" = Whimper$  
Truth$ <> Ugly$  
Happiness$ => "Warm puppy"
```





## Do This

Enter a program that lets you experiment with string comparison.

1. Type in this program:

```
DO
INPUT "First string: "; First$
IF First$ = "stop" THEN EXIT
INPUT "Second string: "; Second$
IF First$ > Second$ THEN PRINT First$; " is greater than "; Second$
IF First$ < Second$ THEN PRINT First$; " is less than "; Second$
IF First$ = Second$ THEN PRINT First$; " is the same as "; Second$
PRINT
LOOP
```

2. Run the program; when it asks you to type in strings, use these exact examples (watch for capital letters):

First String	Second String
a	b
A	B
a	B
bat	bat
bat	cat
battle	cat
rattle	rat
Zambia	ancillary
127	63
5	x
;	g

3. Experiment by typing in your own values for the two strings; see if you can find a pattern emerging.
4. When you're through experimenting, type "stop" to end (by the way, "STOP" won't work!).

### ASCII: How One Word Can Be Less Than Another

While you can compare strings in the same way you compare numbers, BASIC doesn't make the comparisons in the same way. BASIC recognizes numbers as numbers, but it doesn't recognize characters in a string specifically as characters. Rather, it sees each character as a specific code number corresponding to one of the elements of the international coding system called *The American Standard Code for Information Interchange* (ASCII—pronounced “askey”—for short). Table 8-1 shows the printable ASCII codes and their corresponding characters. The numbers on the left represent the ASCII code numbers that generate the characters using the CHR\$ function. Numbers 33 through 126 are the universal ASCII set; numbers 128 through 217 are specific to the Macintosh. Numbers 1 to 32 and number 127 are reserved for special codes that give technical instructions to the computer.

For example, when BASIC looks at an uppercase A, it sees ASCII 65; when it looks at the semicolon character ;, it sees ASCII 59. When BASIC comes across a digit in a string, it doesn't recognize the digit as a number but as an ASCII character, which in turn has its own number. Thus, in the string “1 is the loneliest number”, BASIC sees the character “1” as ASCII 49.

When you ask BASIC to compare strings, it compares ASCII code numbers. For example, when you have BASIC compare the string “5” (ASCII 53) with the string “x” (ASCII 120), it compares 53 with 120; since 53 is less than 120, “5” is less than “x”. BASIC makes decisions about the inequality of strings based on the first nonidentical character. If all the characters are the same, the strings are the same—“bat” and “bat”, for instance.

The string “rattle” is greater than “rat” not because “rattle” is longer, but because “rattle”'s fourth letter is “t”, or ASCII 84; “rat”'s fourth letter doesn't exist to you or to me—but to BASIC, it exists as the **null string character**. To BASIC, the null string character is always less than any other character. Thus BASIC compares 84 to a number that is by definition less than 84 and concludes that “rattle” is greater than “rat”.

Table 8-1 ASCII Character Chart

	56) 8	81) Q	106) j	131) É	156) ú	181) μ	206) Œ
32)	57) 9	82) R	107) k	132) Ñ	157) ù	182) ð	207) œ
33) !	58) :	83) S	108) l	133) Ò	158) û	183) Σ	208) -
34) "	59) ;	84) T	109) m	134) Ù	159) ü	184) Π	209) -
35) #	60) <	85) U	110) n	135) á	160) †	185) π	210) "
36) \$	61) =	86) V	111) o	136) â	161) °	186) ϣ	211) "
37) %	62) >	87) W	112) p	137) ã	162) ‡	187) ρ	212) '
38) &	63) ?	88) X	113) q	138) ä	163) £	188) ρ	213) '.
39) '.	64) @	89) Y	114) r	139) å	164) §	189) Ω	214) ÷
40) (	65) A	90) Z	115) s	140) å	165) ●	190) œ	215) ◇
41) )	66) B	91) [	116) t	141) ç	166) ¶	191) ø	216) ÿ
42) *	67) C	92) \	117) u	142) é	167) β	192) ÷	217) ⚡
43) +	68) D	93) ]	118) v	143) è	168) @	193) i	
44) ,	69) E	94) ^	119) w	144) ê	169) ©	194) ~	
45) -	70) F	95) _	120) x	145) ë	170) ™	195) ✓	
46) .	71) G	96) `	121) y	146) í	171) ´	196) f	
47) /	72) H	97) a	122) z	147) ì	172) ¨	197) ≈	
48) 0	73) I	98) b	123) {	148) î	173) ≠	198) Δ	
49) 1	74) J	99) c	124)	149) ï	174) Æ	199) «	
50) 2	75) K	100) d	125) }	150) ñ	175) Ø	200) »	
51) 3	76) L	101) e	126) ~	151) ó	176) ∞	201) ...	
52) 4	77) M	102) f	127)	152) ò	177) ±	202)	
53) 5	78) N	103) g	128) Ä	153) ô	178) ≤	203) À	
54) 6	79) O	104) h	129) Å	154) ö	179) ≥	204) Ã	
55) 7	80) P	105) i	130) Ç	155) õ	180) ¥	205) Õ	

"Zambia" is less than "ancillary" not because "Zambia" is shorter, but because uppercase letters like Z have lower values in the ASCII chart than lowercase letters like a; BASIC stops comparing when it finds the first difference. That's why entering the word *STOP*, with all uppercase letters—ASCII 83 84 79 80—won't end the program you've just used; BASIC is looking for *stop*—ASCII 115 116 111 112.

By the same rules, the string "127" is less than the string "63" because the initial character in the second string is ASCII code number 49, whereas the initial character in the second string is ASCII code number 54. Remember, as far as BASIC knows, these aren't numbers; they are characters in a string.



## String Literals vs. String Variables

BASIC compares string literals to each other. When you ask BASIC to compare string variables, it looks at the variables' *contents*. BASIC never compares string variable *names*.

```
Author$ = "Jean-Paul Marat"  
Critic$ = "Charlotte Corday"  
IF Author$ > Critic$ THEN PRINT "Marat" ELSE PRINT "Corday"
```

Don't confuse a name  
with what it represents.

Running this program produces the output "Marat". The program compares the contents of variable Author\$ with the contents of variable Critic\$. The first character of Author\$'s contents is *J*; the first character of the contents of Critic\$ is *C*. *J* is ASCII 74, which is compared to ASCII 67 for *C*. Therefore, to BASIC, Author\$ is greater than Critic\$. If BASIC were comparing string variable names, the *C* in Critic\$ would be greater than the *A* in Author\$ (ASCII 65), making Critic\$ greater than Author\$.



### For Your Information

**Computer Logic** The way that BASIC compares strings is an excellent example of the way computer thinking differs from human thinking. If you want to communicate effectively with your computer, you have to think as it does. It's not that computer logic is so different from human logic; it's just that computers work with different premises.

Here's a summary of the rules for comparing strings with BASIC:

- All comparisons are based on ASCII code numbers.
- For comparison purposes, BASIC looks for the first non-identical character in the two strings.
- Strings with exactly the same characters in exactly the same positions are considered equal.



- If the initial characters of two strings are exactly the same and in the same positions but one string is longer, the longer string is considered greater. In all other cases, the length of a string has no bearing on string comparison.
- BASIC always makes comparisons between string literals. When BASIC is asked to compare two string variables or a string variable and a string literal, it first converts the variable to the string literal that it represents.

### Getting to ASCII: The Function ASC

You'll sometimes run across situations when you want to know the ASCII value of a particular character. That's likely to be the case when you know the name of a variable but not the variable's contents.

The function **ASC** returns the ASCII code for the first character in a given string. Here's how it works in a couple of BASIC statements, using both a string literal and a string variable:

```
PRINT ASC("The play's the thing")  
  
Whatever$ = "23 Skidoo!"      ! Time warp  
Asciinum = ASC(Whatever$)
```

The first example displays the number 84, the ASCII code for uppercase *T*. The second example assigns the ASCII code number for the first character of *Whatever\$*'s contents to the numeric variable *Asciinum* (it happens to be 50).

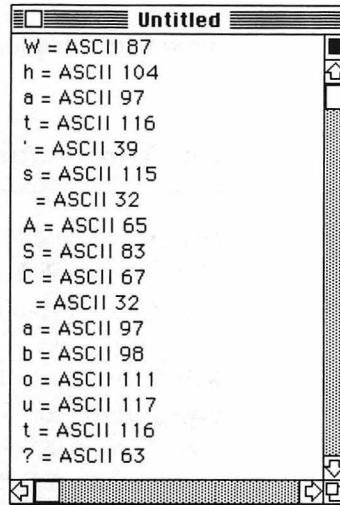


### Do This

Enter and run this program, showing the ASCII values for characters in a given string:

```
Sample$ = "What's ASC about?"  
  
FOR Numbers = 1 TO LEN(Sample$)  
    Character$ = MID$(Sample$, Numbers, 1)  
    PRINT Character$; " = ASCII "; ASC(Character$)  
NEXT Numbers
```

Figure 8-6 shows what should appear in your output window.



**Figure 8-6** “What’s ASC( about?”

The functions in that last example flew thick and fast, so here’s a line-by-line explanation:

`Sample$ = “What’s ASC about?”` Assign the string literal “What’s ASC about?” to the string variable `Sample$`.

`FOR Numbers = 1 TO LEN(Sample$)` Set up a `FOR\NEXT` loop to repeat as many times as there are characters in the contents of `Sample$`.

`Character$ = MID$(Sample$, Numbers, 1)` Let the variable `Character$` hold one character from the contents of `Sample$`. The particular character depends on the current value of the numeric, variable `Numbers`—if `Numbers` holds 3, assign the third character; if `Numbers` holds 5, assign the fifth character; and so on.

`PRINT Character$; “= ASCII ”; ASC(Character$)` Display a line of text beginning with the character assigned to `Character$`, followed by the string “= ASCII ”, and concluding with the ASCII code number of the character assigned to `Character$`.

**NEXT Numbers** Repeat the process until all the characters assigned to Sample\$ have been accounted for.



### For Your Information

**Phantom ASCII** Sometimes you don't know how many characters are in a given string, and you might find yourself looking for the value of a character that doesn't really exist—as in the case of a variable that hasn't any characters assigned to it (that is, the null string). If you ask BASIC to give you the ASCII value for the null string, it gives you -1. There's no value for the null string in the official ASCII code, so Macintosh BASIC creates its own.

This little anomaly can be useful. Let's say you wanted to use a DO\LOOP construction instead of FOR\NEXT in the previous example:

```
Numbers = 0
Sample$ = "What's ASC about?"
DO
    Numbers = Numbers + 1
    Character$ = MID$(Sample$, Numbers, 1)
    IF ASC(Character$) = -1 THEN EXIT
    PRINT Character$; " = ASCII "; ASC(Character$)
LOOP
PRINT CHR$(217)           ! You haven't seen this yet
```

See if you can figure out what PRINT CHR\$(217) is about before you go on.

### CHR\$—Producing Characters from ASCII Code Numbers

The last sample program you ran showed you that ASCII 32 is the space character. If you want to assign that character to a string variable (call it Space\$), you could handle it like any other character: type a quote mark, press the space bar once, and type another quote mark, like this:

```
Space$ = " "
```



That's the way you've done it up to now. An alternate way to do the same thing is to assign the character's ASCII value to a string, using the function **CHR\$**. **CHR\$** is the flip side of **ASC**; given the ASCII number, **CHR\$** returns the character.



### Do This

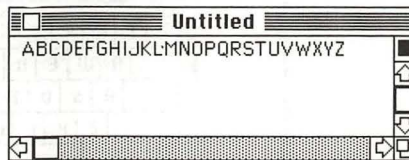
Type in this program to show how **CHR\$** works.

```
A = 65
Z = 90

FOR Uppercase.Letter = A TO Z
    Letter$ = CHR$(Uppercase.Letter)
    PRINT Letter$;
NEXT Uppercase.Letter
```

Figure 8-7 shows your output.

A glance back at the ASCII codes in Table 8-1 shows that the uppercase letters occupy numbers 65 through 90. **PRINT CHR\$(65)** produces the same result as **PRINT "A"**. A statement like **PRINT CHR\$(217)** produces a result you just can't duplicate from the keyboard.



**Figure 8-7** **CHR\$** alphabet



## MacBASIC's ASCII Extensions

Standard ASCII defines code numbers 0 through 127. The first 32 codes and code number 127 don't produce text characters; these numbers are reserved for special codes that give technical instructions to the computer. You don't have to worry about them at this stage of your programming career. There are also 128 unused code numbers (128 through 255); MacBASIC gets use out of numbers 128 through 217 by assigning them special text symbols and non-Roman letters. You can use any of these characters by assigning them to strings through the CHR\$ function.



### Pop Quiz

#### Question 2

Write a program to display the code number and corresponding MacBASIC special text character for each of codes 128 through 217.

Choose Key Caps from this menu.

You can select and cut this for later pasting into a BASIC program.

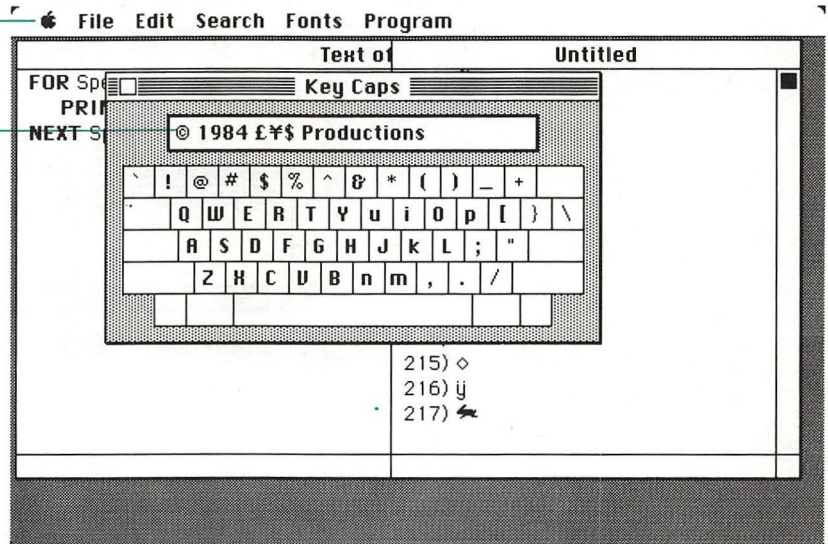


Figure 8-8 Getting characters from Key Caps

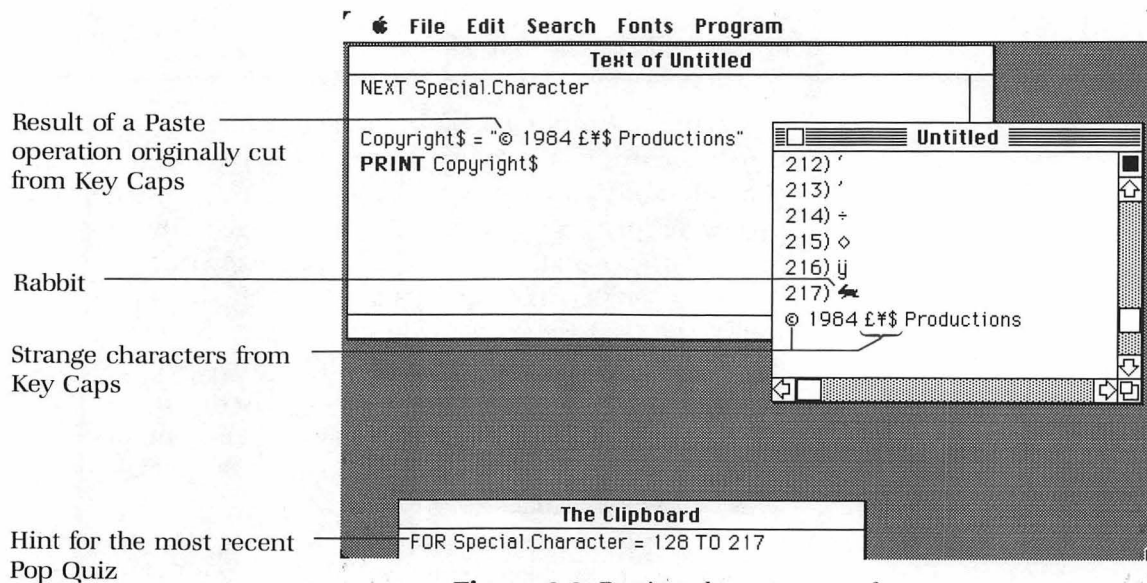


## For Your Information

**Key Caps to Construct Strings** Sometimes you want to construct strings made up of both common and unusual characters, but you don't want to go to the trouble of looking up the proper ASCII codes in order to use CHR\$. That's particularly true if there are a *lot* of special characters; who wants to retype CHR\$ 20 times? For such situations the **Key Caps** desk accessory can come in really handy. As Figure 8-8 shows, Key Caps lets you see the special characters that the keyboard produces when you press and hold down the Option key (or the Option key in combination with the Shift key), which makes most of the special characters directly available from the keyboard.

The Option key actually gets you the special characters; Key Caps lets you see what you'll get, since the special characters don't appear on the surface of the keyboard. As you type or click on the Key Caps with the mouse, the appropriate characters appear in the Key Caps typing box. You can then select any or all characters in the box to cut to the Clipboard and paste to your BASIC program, as shown in Figure 8-9. Go and experiment with it for a while!

You'll still have to use CHR\$(217) to get the rabbit, though.



**Figure 8-9** Pasting characters cut from Key Caps

### Make Your Own ASCII Chart

Here are the programs I used to generate all the ASCII characters for Table 8-1. The first one generates the standard characters:

```
FOR Ascii = 33 TO 126
  PRINT Ascii; " "; CHR$(Ascii)
NEXT Ascii
```

Here's what I used to get the special Macintosh characters:

```
FOR Ascii = 128 TO 217
  PRINT Ascii; " "; CHR$(Ascii)
NEXT Ascii
```

---

**TIME\$ and DATE\$**

---

BASIC provides two handy system functions for reporting the current time and date. Appropriately called **TIME\$** and **DATE\$**, these two functions take no arguments. In North America **TIME\$** gives the hour, minute, and second in the form HH:MM:SS, with HH in the range 1 through 12, MM in the range 0 through 59, and SS in the range 0 through 59. At the end of the string comes a space and then either AM or PM. Again in North America, **DATE\$** gives the month, day, and year in the form MM/DD/YY with MM in the range 1 through 12, DD in the range 1 through 31 (adjusting automatically for short months and leap year Februaries), and YY in the range 00 through 99. An example of **TIME\$** and **DATE\$** is shown in Figure 8-10.



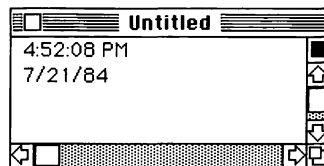
---

**Do This**

Type in this two-line program to display the current time and date:

```
PRINT TIME$  
PRINT DATE$
```

You'll probably find yourself using the time and date functions often. Any business, education, or science programs producing reports need at least the date; reports produced several times daily also require the time to keep the various versions from being mixed up.



**Figure 8-10** TIME\$ and DATE\$





## Pop Quiz

### 5000-Point Bonus Question (Optional)

Tackle this one only if you think you really know your stuff! Write a program that announces the correct time every 10 seconds. The program must work correctly for at least seven announcements (that is, covering a period of at least 70 seconds). If you get this one, pick up your Programmer's Hero Badge.

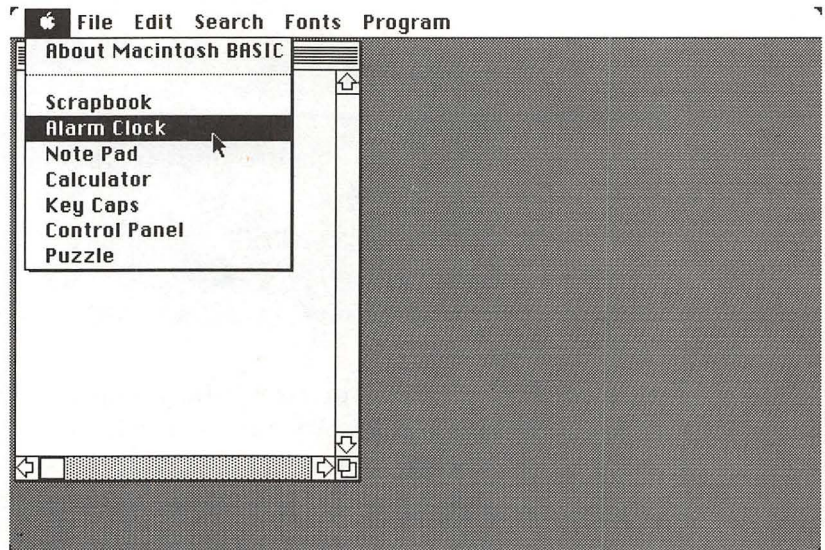


Figure 8-11 Summoning the Alarm Clock



## For Your Information

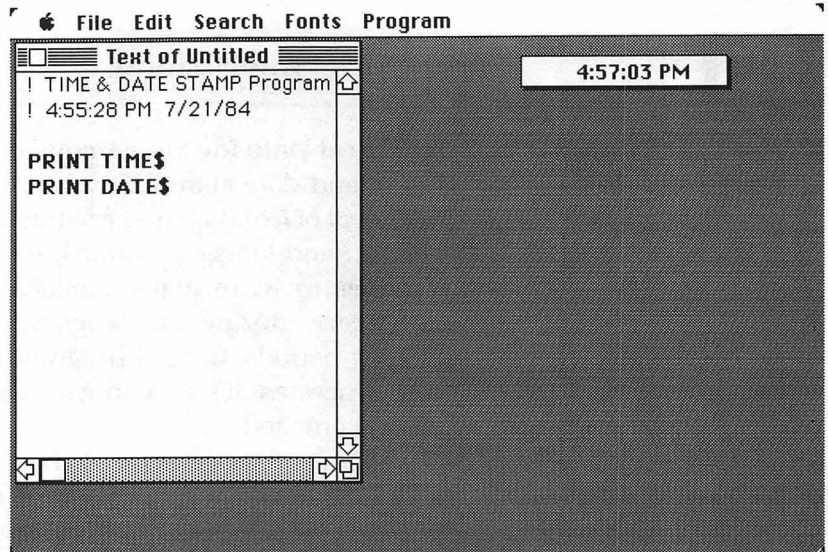
**Time and Date for Your Program Versions** Including a time and date stamp on your programs will save you a great deal of irritation and frustration when you write more complex and longer programs. It's not unusual for a programmer to write many versions of the same program, particularly during the program's development and debugging periods. Careful programmers also save copies of their programs. It's easy to get these various versions and copies confused.

But using TIMES\$ and DATE\$ isn't the answer. These functions provide the *current* time and date, and you want the time and date of the program's *creation*. A good solution is to do a cut-and-paste operation, using the **Alarm Clock** desk accessory to get the information and the comment marker character to store it in your program. Assuming the program you want to mark with the time and date is already on your desktop, here's how to do it.

1. Choose Alarm Clock from the Desk Accessory menu, as shown in Figure 8-11.
2. Choose Cut from the Edit menu (or enter ⌘ X); the time and date are automatically selected.
3. Enter the comment marker character (!) where you want the time and date to appear; near the start of the program is best, just below the program name.
4. Leave the insertion point just after the comment marker character; then choose Paste from the Edit menu (or enter ⌘ V).

Figure 8-12 shows you the end result.





**Figure 8-12** Using the Alarm Clock to date-stamp a program

## Summary

### New Terms

**ASCII** acronym for *American Standard Code for Information Interchange*, a coding system used by Macintosh BASIC to represent all text characters the machine is capable of reproducing.

**Concatenation** Operation to combine two or more strings. The concatenation operator is the ampersand character, &.

**Null string character** theoretical character that returns an ASCII code of -1 when it appears as the argument to the function ASC.

**Substring** part of a string returned through one of the substring functions (LEFT\$, MID\$, RIGHT\$).

### Menu Items

**Alarm Clock** desk accessory showing time and date. You can cut or copy the time; later you can paste it into a BASIC program by assigning it to a string or by displaying it after the comment marker character.

**Key Caps** desk accessory showing characters produced by various keys, including the "hidden" character set that you get by holding down the Option key. You can cut and copy characters typed to Key Caps's box and later paste them into a BASIC program.

---

## Programming Statements and Characters

---

**&** string concatenation operator; used to combine two or more strings into one.

**ASC** numeric function taking a string argument (literal, variable, or expression) and returning the ASCII code for the first character in the argument.

**CHR\$** string function taking a numeric argument (constant, variable, or expression) in the range 0 through 255 and returning the ASCII character or MacBASIC special character represented by the argument.

**DATE\$** system string function taking no argument and returning the current date.

**LEFT\$** string function taking two arguments (a string and a numeric) and returning as many characters from the start of the first argument's string as is dictated by the second argument's value.

**LEN** numeric function taking one string argument and returning the number of characters in the argument.

**MID\$** string function taking two arguments (a string and a numeric) or three arguments (a string and two numerics) and returning as many characters from the first argument's string as is dictated by the other arguments: when one numeric is used, all characters from the numeric's position to the last character in the string are returned; when two numerics are used, as many characters as stipulated by the second numeric are returned beginning with the character whose position is stipulated by the first numeric.

**RIGHT\$** string function taking two arguments (a string and a numeric) and returning as many characters from the end of the first argument's string as is dictated by the second argument's value.

**TIME\$** system string function taking no argument and returning the current time.

---

## Pop Quiz Answers

---

### Question 1

```
PRINT RIGHT$(Name$, LEN(Name$) - Search)
```

### Question 2

```
FOR Special.Character = 128 TO 217
  PRINT Special.Character; " "; CHR$(Special.Character)
NEXT Special.Character
```

### Bonus Question

My code for the 5000-Point Bonus Question follows.



```

Limit = 10                                ! Check every 10 seconds
Current.Time$ = TIME$                    ! What time is it now?
PRINT "Current time: "; Current.Time$

FOR Repeat = 1 TO 7                      ! Do this seven times
    DO                                    ! Loop until 10 seconds gone
        IF Current.Time$ < TIME$ THEN GOSUB Clock: ! Has a second passed yet?
        IF Times.Up = 1 THEN EXIT          ! 10 seconds gone?
    LOOP                                  ! Back to the top

    PRINT "Current time: "; TIME$
    Times.Up = 0                          ! Reset timer
    Seconds = 0                           ! Reset second-counter

NEXT Repeat                              ! Do it again

PRINT "There you go - 70 seconds!"
END PROGRAM

CLOCK:
    Current.Time$ = TIME$                ! Update the correct time
    Seconds = Seconds + 1                ! Update the second-counter
    IF Seconds = Limit THEN Times.Up = 1 ! If 10 seconds gone
    RETURN                               ! then our time's up

```

## Commands, Menu Items, Keywords You Know So Far

Commands and Menu Items		Programming Statements	
Alarm Clock	Open	ASC	INVERT
Backspace Key	Option key	BTNWAIT	LEFT\$
Clear	Paste	CHR\$	LEN
Copy	Run	CLEARWINDOW	MID\$
Cut	Save	DATE\$	MOUSEB
Halt	Select All	DO\LOOP	MOUSEH
Key Caps	Undo	ELSE	MOUSEV
New	⌘ key	END PROGRAM	OVAL
<b>Programming Characters</b>		EXIT	PAINT
+	-	FOR...TO...STEP\NEXT	PLOT
*	=	FRAME	RECT
<	>	GOSUB	RETURN
\$	!	IF...THEN	RIGHT\$
		INPUT	RND
		INT	TIME\$

---

**Bughouse**

---

The following absurdly inelegant program is meant to print two lines on the screen. The first is supposed to say "BASIC is a snap." The next tab field over on the same line the ASCII value for the period is supposed to appear. The second line is supposed to say "BASIC", followed by a tab field and the ASCII code for uppercase B. The code you see would print that text (convoluted though it be), if it weren't for six pesky bugs.

```
Code$ = "Learning how to program in BASIC is a snap."
```

```
FOR Find.B = 1 TO LENGTH(Code$)
```

```
  IF MIDDLE$(Code$, Find.C, 1) = "B" THEN EXIT
```

```
NEXT Find.B
```

```
Code$ = RIGHT$(Code$, LEN(Code$) - (Find.B - 1))
```

```
BASIC$ = LEFT$(Code$, 5, 1)
```

```
PRINT Code$, CHR$(Right(Code$, 1))
```

```
PRINT BASIC$, ASC(Code$)
```

# SESSION

## 9

## Arrays and Other Data

---

**T**his session's major subject matter is variables, yet again. But the variables you'll learn about in this session are of a different type than the ones you've seen before; **array variables** are special in that they let you store a number of different values under the same name. You'll also read about a set of keywords, `READ\DATA\RESTORE`, that let you keep information in lists of program lines; they give you the power to assign values to variables only when you want to. In addition to these major matters, you'll get some experience using the Calculator desk accessory and some of the commands from the Search menu.

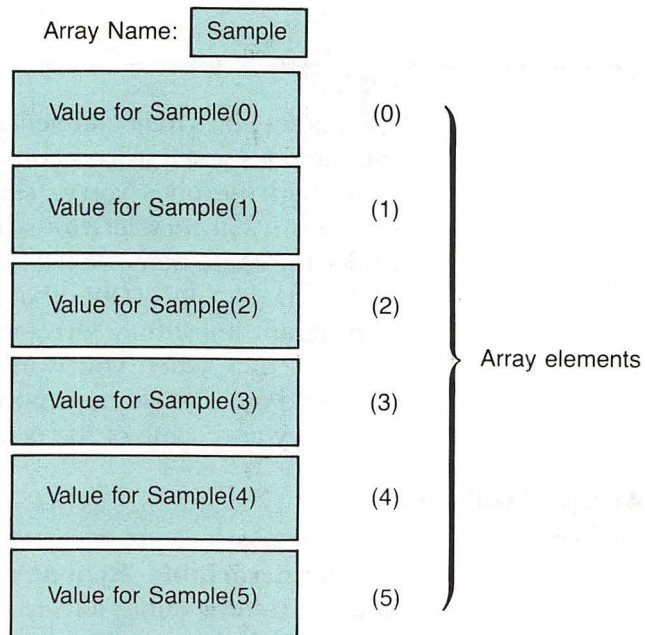
### What Arrays Look Like

---

The term **variable structure** refers to the way a computer language stores values for variables. Although you don't need to know the technical details, you do need to understand that Macintosh BASIC has two kinds of variable structures, simple and array.

Up to now, you've been using only simple variables. Each simple variable has its own variable name, and only one value can be assigned to it at a time. An **array** variable, on the other hand, is like a collection of variables with the same name. Each variable in the collection is distinguished from the others by a unique **numeric subscript**—a numeric constant, variable, or expression appearing in parentheses following the array name. The individual units of the array are called **elements**.

Figure 9-1 shows the structure of an array variable called Sample. This particular array has six elements in it, which means that you can assign it six different values at the same time. Note that the lowest numbered element is number 0. (Computers count funny—they start from 0 instead of from 1; but what can you expect from a silicon-based life form?)



**Figure 9-1** Skeletal array structure



Both the name and the number of elements are arbitrary; I could just as well have called the array variable *Ralph* and given it 1000 elements. I chose to call it *Sample*, which means that its elements must be called respectively *Sample(0)*, *Sample(1)*, *Sample(2)*, *Sample(3)*, *Sample(4)*, and *Sample(5)*. The names for all the elements are the same—in this case, *Sample*. You distinguish the individual elements by their numeric subscripts, the numbers that appear in parentheses. Element number 0 is *Sample(0)*, element number 4 is *Sample(4)*, and so on. The boxes in Figure 9-1 represent the values that the elements hold.

You assign values to array variable elements the same way you do to simple variables—for instance, you type

```
Sample(3) = 30
```

to assign the value 30 to element number 3 of numeric array *Sample*. But before you can give any array element a value, you have to give BASIC some preliminary information.

### **DIM—Telling BASIC How to Set Up the Array**

The keyword **DIM** is short for dimension, meaning size or scope. Before you can use an array, you have to tell BASIC the array's size so that it can set aside enough memory space for all the variable elements.



### Do This

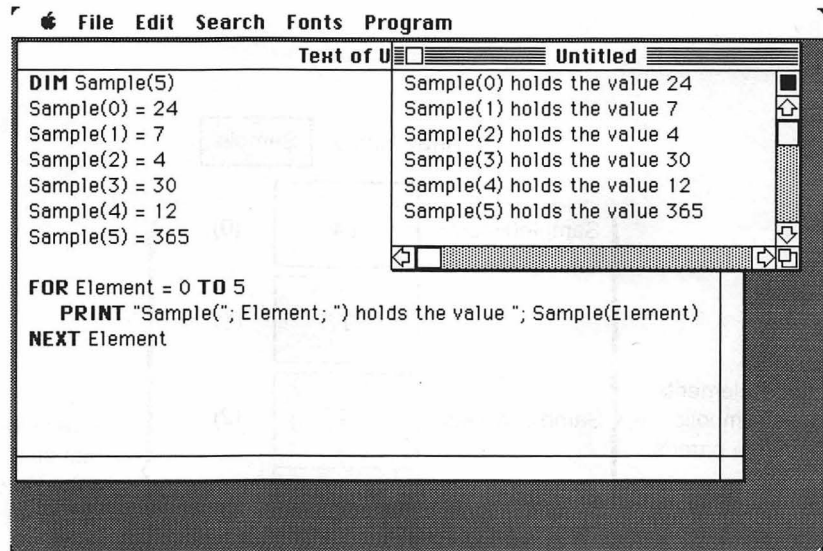
Type in and run the following program, which sets up a numeric array with six elements. Don't type any spaces between the last character in the array name and its open parenthesis. As usual, try to figure out exactly what the program will do before you run it.

```
DIM Sample(5)
Sample(0) = 24
Sample(1) = 7
Sample(2) = 4
Sample(3) = 30
Sample(4) = 12
Sample(5) = 365

FOR Element = 0 TO 5
  PRINT "Sample("; Element; ") holds the value "; Sample(Element)
NEXT Element
```

Figure 9-2 shows you what your output window should look like.

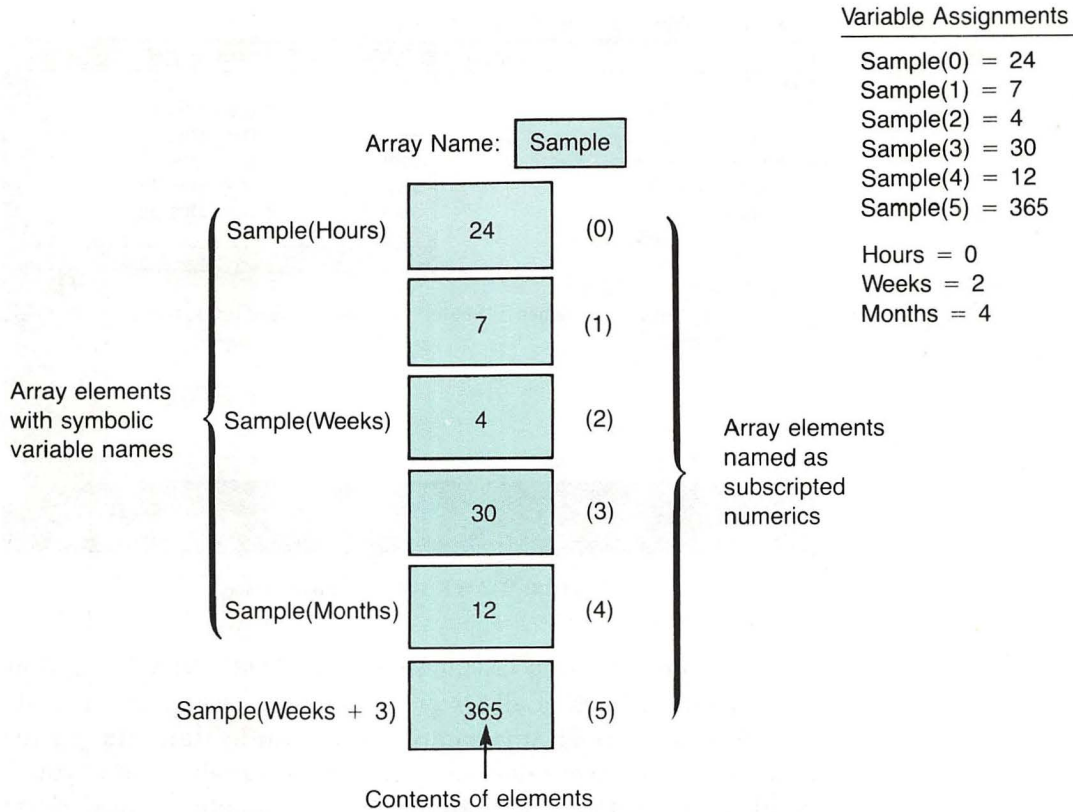
Here's how the program works. The first group of statements sets up the array [DIM Sample(5)] and assigns values to the various elements. Because the computer starts counting with 0, your DIM statement sets up space for a total of six array elements even though you gave it the number 5. You can always count on the computer to give you one more element than you ask for.



**Figure 9-2** Output of First Array

The second group of statements is a `FOR\NEXT` loop that displays the values of all six array elements. Pay particular attention to it; through it you can begin to understand the power of arrays. If you were dealing with simple variables here, you'd need six separate `PRINT` statements to duplicate the work done by this one loop-enclosed statement. Each time through the loop the value of `Element` increments, from 0 through 5, referring each time to a different subscripted numeric (that is, numbered array element). When the loop starts and the variable `Element` holds 0, the `PRINT` statement shows the value of element 0, the value 24. On the next pass, `Element` holds 1, so `PRINT` displays the value stored in element 1, the value 7; and so it goes for the other elements. Figure 9-3 shows what's going on.

Save this program under the name *First Array*; you'll need it later when you learn how to use some of the Search Menu commands.



**Figure 9-3** Skeletal array structure showing values of elements



### For Your Information

**Element 0 Is a Bonus** You don't need to use the 0th element of an array. Many programmers find it less confusing to begin with element number 1 and ignore element 0 altogether. Sometimes a programmer uses element 0 as a counter or as a handy place to store a running total to be used later for some other purpose. Element 0 of an array is like any other variable; you can use it for whatever you want.





## Pop Quiz

### Question 1

Write a program that prompts the user for 10 values, stores the values in an array called Quiz, and then displays the 10 values. Don't use element 0 to store any of the values. 500 bonus points and Budding BASIC Genius Award if you can figure out how to store the sum of the values in element 0.

## Using Variables to Reference Array Elements

You already know that you can almost always use a variable or an expression in BASIC in place of a number. The PRINT statement in the FOR\NEXT loop of the last example shows you how to **reference** (that is, refer to) an array element using a variable name instead of a number

More astute readers (and those who are compulsive about figuring out obscure coding systems) may have realized that the values in the array called Sample all refer to time periods: element 0 holds the number of hours in the day, element 2 holds the weeks in a month, and so on. But names have more meaning to most people than numbers do; and luckily, BASIC lets you use variable names for array elements. The next example shows you how to do it.

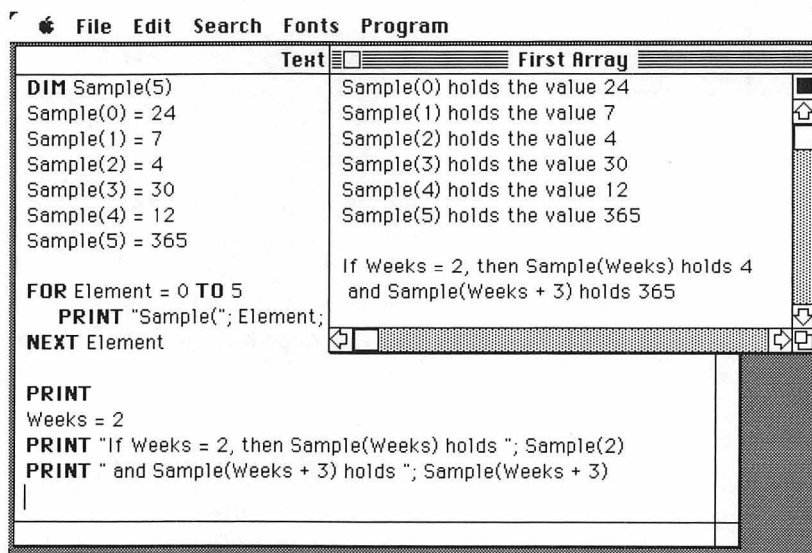


## Do This

Add these lines to the end of the preceding program and run it.

```
PRINT
Weeks = 2
PRINT "If Weeks = 2, then Sample(Weeks) holds "; Sample(Weeks)
PRINT " and Sample(Weeks + 3) holds "; Sample(Weeks + 3)
```

Figure 9-4 shows you what the output looks like.



**Figure 9-4** Using symbolic names for array elements

The final PRINT statement in this example ought to tweak your interest; it shows an expression—`Sample(Weeks + 3)`—used to reference an array element (how's that for a jargony sentence!). Saying `Weeks + 3` is the same as saying `2 + 3` because the variable `Weeks` holds the value 2. So `Sample(Weeks + 3)` references `Sample(5)`.

### Array Arithmetic

You can do arithmetic with array variables just as you can with simple variables. The following "Do This" box gives you a simple exercise that lets you see array arithmetic in action and suggests that you use the **Calculator** desk accessory to help you predict the program's output. This calculator works like the simple four-function arithmetic calculator you've probably used a million times before; most likely you won't need instructions for using it. If you do, you'll find them in your *Macintosh* owner's guide.



## Do This

Type in and run this short program demonstrating array arithmetic.

1. Type in this code:

```
DIM Numbers(5)
Numbers(1) = 250
Numbers(2) = 135
Numbers(3) = Numbers(1) - Numbers(2)
Numbers(4) = Numbers(3) / Numbers(1)
Numbers(5) = Numbers(4) * Numbers(3)

FOR Values = 1 TO 5
    PRINT "Numbers("; Values; ") = "; Numbers(Values)
NEXT Values
```

2. Figure out what the program will do before you execute it. *Please don't skip this step!* It's especially important that you analyze what each variable holds. Use the Calculator desk accessory to help you.
3. Execute the program.

Figure 9-5, on the next page, shows what you should get.

Here's an explanation of the three lines that do arithmetic:

$\text{Numbers}(3) = \text{Numbers}(1) - \text{Numbers}(2)$

is the same as

$\text{Numbers}(3) = 250 - 135$

which comes out to 115. By the same logic,

$\text{Numbers}(4) = \text{Numbers}(3) / \text{Numbers}(1)$

is the same as

$\text{Numbers}(4) = 115 / 250$

which is .46. And finally,

$\text{Numbers}(5) = \text{Numbers}(4) * \text{Numbers}(3)$

is another way of saying

$\text{Numbers}(5) = .46 * 115$

which is 52.9.

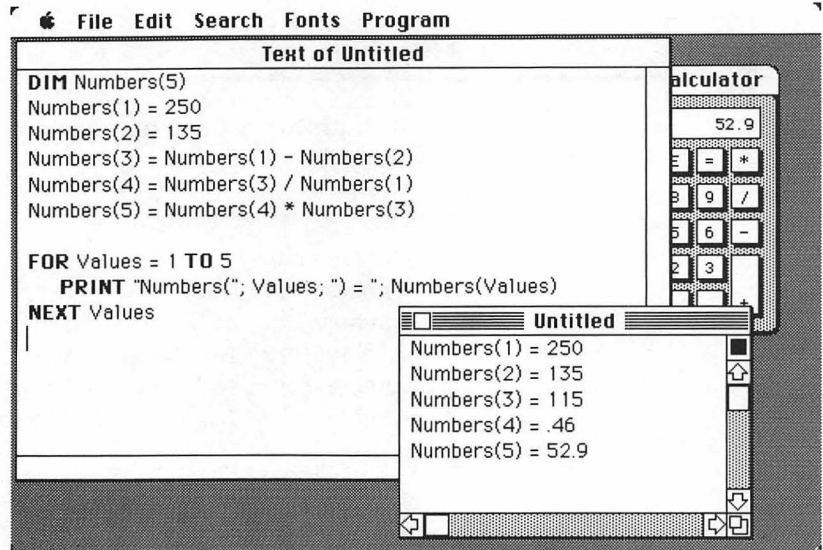


Figure 9-5 Array arithmetic



### For Your Information

**Functions Work with Arrays** Since arrays are just like other variables (except better), you can use functions with them. For instance, assuming that `Numbers(5)` from the last example still holds 52.9, the statement `PRINT INT(Numbers(5))` results in 52. Just remember to include all the parentheses!



Spend some time experimenting with arrays and array arithmetic. Make mistakes—they don't count against you. Use numeric functions with your arrays. Your stock of BASIC keywords is pretty large now, so use them to make up some programs of your own. Then go on to read about string arrays.

### **Arrays for Strings**

String arrays look just like numeric arrays, except that you need the string marker character \$ at the end of the array name before the open parenthesis. This next exercise lets you see how a string array works and also gives you some experience using the Search menu.

**Setting Search and Replace Options** If you pull down the Search menu, you'll see that there are four items on it: Find, Replace, Replace All, and What to Find. The first three items depend on the fourth, so it makes sense to look at that one first. You're going to replace all instances of the variable name Sample with the variable name Sample\$.



## Do This

Change the *First Array* program to learn about string arrays.

1. Open the *First Array* program by entering ⌘O and double-clicking *First Array* from the list presented.
2. Select and clear the four lines below the FOR\NEXT loop.
3. Select **What to Find** from the Search menu. This produces the What to Find dialog box.
4. In the box labeled **Search for**, type the word *Sample*, as shown in Figure 9-6.
5. Move the insertion point to the **Replace with** box.
6. Type the word *Sample\$* and either click OK or press Return. (Ignore the other items in the box for now. I'll explain them in a minute.)
7. Choose **Replace All** from the Search menu to replace all occurrences of *Sample*.
8. Replace the numbers 24, 7, 4, 20, 12 and 365 with the quoted strings as they appear in the listing below. (Make these changes with the mouse rather than with the Search commands; it's quicker.)
9. Repeat steps 2 through 5 and choose Replace to replace the word *value* with the word *name* (just for the practice). Here's what you should end up with:

```
DIM Sample$(5)
Sample$(0) = "Horatio"
Sample$(1) = "Elenore"
Sample$(2) = "Duncan"
Sample$(3) = "Othello"
Sample$(4) = "Desdemona"
Sample$(5) = "Fido"
```

```
FOR Element = 0 TO 5
```

```
  PRINT "Sample$(""; Element; ") holds the name "; Sample$(Element)
```

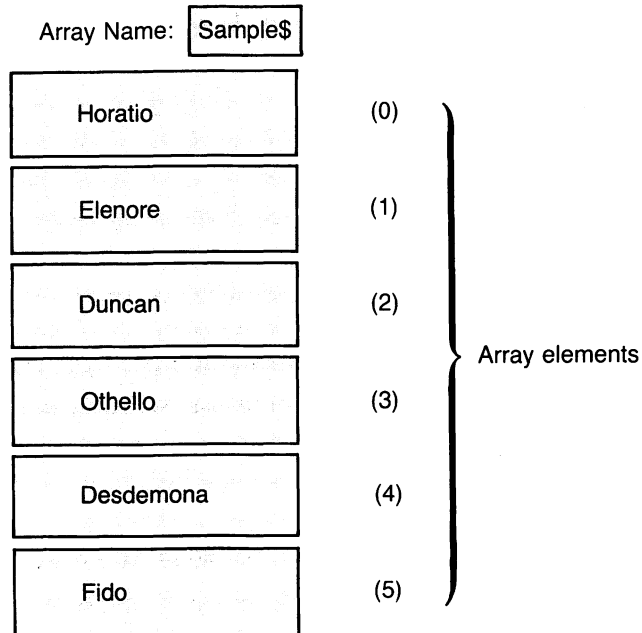
```
NEXT Element
```

10. Predict what will happen and then run the program.

Figure 9-7 shows you what's happening in the computer's array memory.

**Figure 9-6** What to Find dialog box

**A Few Words About the What to Find Dialog Box** There are two additional pairs of items in the What to Find box: Separate Words and Include Embedded Words, and Ignore Case and Match Case (*Case* here means upper and lower case, as in capital and small letters).



**Figure 9-7** Skeletal array for Sample\$

**Separate Words** means “Look for instances of the word(s) in the ‘Search for’ box that stand alone.” If you click the button next to **Include Embedded Words**, BASIC looks for instances of the “Search for” word(s) that either stand on their own or are parts of other words. For instance, if you tell BASIC to search for the word *to* and then click the Include Embedded Words box, it would find *to* in words like *toward* and *astounded*.

The **Ignore Case** button is pretty clear, I think. If you click in **Match Case**, BASIC will find Ant in *Antfarm* but not in *Rant* and *Rave*. The real way to understand this stuff, of course, is to experiment with it.

**Save a Copy In...—Storing a Program with a New Name** If you save the program on a disk now, without changing the program’s name, BASIC will replace your old numeric array program with it. Assuming that you want to keep the original version of the program intact and that you want to save this new version under a more appropriate name, such as *First String Array*, you can do one of two things. The first is to use Select All and Copy to put the whole program onto the Clipboard, choose New and then Paste to move the program into a new listing window, and finally choose Save Text, giving the program the new name *First String Array* (those of you who like shortcuts can keep the Command key pressed down and type ACNVS). The second is to choose Save a Copy In... from the Program menu. Try the first method later; right now, use Save a Copy In.... The computer will tell you what you need to do.

## Functions with String Arrays

All the operators, string functions, and string-related numeric functions you’ve learned so far work with string array variables, just as the numeric functions work with numeric array variables. For example:

```
Love$ = Sample$(3) & “ just dies for ” & Sample$(4)
```

But the typing can get pretty complex; you have to be really careful about matching up all the pairs of parentheses. For instance:

```
News$ = MID$(Sample$(4), 4, 5)           ! Returns demon
Number = ASC(MID$(Sample$(5), 2, 1))      ! Returns 105
```



If all the parenthesis matching becomes too much for you, which wouldn't be surprising, you can always break a complicated statement down into sections. For instance, you can write the complex statement

```
Number = ASC(MID$(Sample$(5), 2, 1))
```

as

```
Temporary$ = Sample$(5)  
Temp2$ = MID$(Temporary$, 2, 1)  
Number = ASC(Temp2$)
```

Spend a few minutes now mixing string array elements with string and string-related functions. Don't go on until you do; it's really important that you learn about keeping your parentheses straight while you're still new to BASIC.



## Pop Quiz

### Question 2

First, figure out what the following totally irrational program does. Then enter it into your computer and run it.

```
DIM Character$(10)  
Character$(0) = ""  
FOR String = 1 TO 10  
    Character$(String) = CHR$(INT(RND(90)) + 33)  
    Character$(0) = Character$(0) & Character$(String)  
NEXT String  
  
FOR Result = 0 TO 10  
    PRINT Character$(Result)  
NEXT Result
```

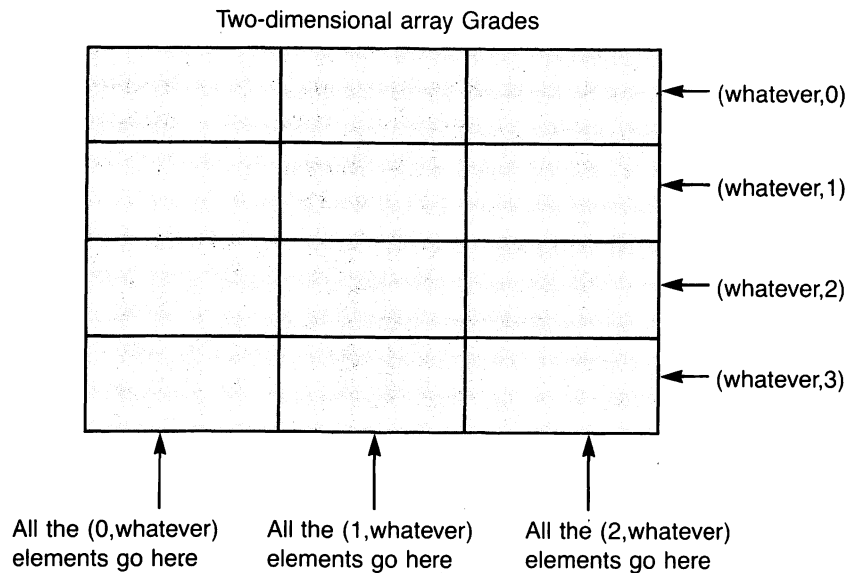
If you predicted what it actually did, hire yourself out as a BASIC consultant. Otherwise, see if you can figure out from the output what this program is about. You may need to run it four or five times (that's OK; no penalty). As a last resort, go to the explanation at the end of the chapter.

## Multidimensional Arrays

So far you've dealt with one-dimensional arrays, which allow you to store a single list of items under the same group name. BASIC also lets you use **multidimensional arrays** for your numbers and strings. These are arrays with multiple lists referred to by the same group name.

The statement `DIM Grades(2,3)` sets up a two-dimensional array with the structure shown in Figure 9-8. The structure of this array has dimensions of width and depth. In the figure, the first of the two dimensions goes from left to right, and the second goes from top to bottom. Lest you be confused by the 0th elements, move right into the next example; the experience of working with an array will give you far more understanding than will reading my ravings.

The statement `DIM Grades(2,3)` produces this



**Figure 9-8** Two-dimensional array



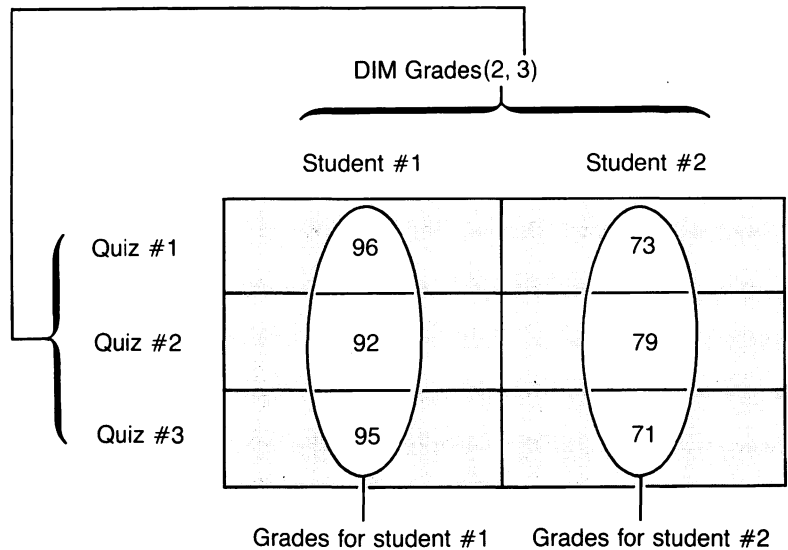
## Do This

Type in and run this program to make understanding multidimensional arrays easier.

1. Get a new listing window and type in the following code:

```
Student.Count = 2
Quiz.Count = 3
DIM Grades(Student.Count, Quiz.Count)
FOR This.Student = 1 TO Student.Count
    PRINT "Enter the grades for student #"; This.Student
    FOR This.Quiz = 1 TO Quiz.Count
        PRINT "Quiz "; This.Quiz;
        INPUT " "; Grades(This.Student, This.Quiz)
    NEXT This.Quiz
    PRINT
NEXT This.Student
BTNWAIT
PRINT Grades(1, 3)
```

2. Immediately save the program under the name *Grades*.
3. Run the program and, in response to the various prompts, enter these grades in the order given: 96, 92, 95, 73, 79, 71.
4. Look at Figure 9-9; it shows you what's going on.
5. Predict what will happen when you press the mouse button.
6. Press the button on the mouse—was your prediction right?



Student #1's grades: 96, 92, 95

Student #2's grades: 73, 79, 71

Note that this example shows only the elements you're using.  
The 0 elements don't appear here (there are six of them).

**Figure 9-9** Two-dimensional array with grades filled in

Your program produces a two-dimensional array that, in effect, establishes several lists, each of which has its own sublist. Ignoring the 0 elements for now, your program sets up a record book for two students, student #1 and student #2. When you run the program, you feed in the results of three quizzes taken by each student. The results of student #1's three quizzes get stored in elements (1,1), (1,2), and (1,3); student #2's grades go into elements (2,1), (2,2), and (2,3).

You can change this program to accept more students taking more quizzes just by changing the values for Student.Count and Quiz.Count. For example, if you had a class of 25 students taking 12 quizzes each, then you'd set Student.Count to 25 and Quiz.Count to 12. Use the 0th elements to hold totals, class and student averages, and so on. In fact, figuring out how to do that would make a great pop quiz!





## Pop Quiz

### Question 3

This one is tough, but you can do it! Change the *Grades* program to store the class average for each quiz and the grade average for each student in the appropriate 0th elements. Make the program display these averages. Use Figures 9-8 and 9-9 to help you figure out what goes where.



## For Your Information

**Even More Dimensions** You can have more than two dimensions in an array. In fact, you can have dozens if you want (although two is enough for most applications); just separate each dimension's range from the next with a comma. The statement `DIM Lots.A.Lists(5, 2, 3, 7)` sets up a four-dimensional array that, counting the 0th elements, gives you six main lists with three sublists for each, four sublists for each of those three, and eight sublists for each of those four. That's 6 times 3 times 4 times 8 separate elements, or 576 in all. To refer to a specific element, you'd need to give four numbers—for example,

```
PRINT Lots.A.Lists(2, 5, 7, 12).
```

Play with multidimensional arrays for a while. Don't be afraid of error messages; if you get one that says "Out of Memory" or "Dimension Too Big", just use smaller numbers in your DIM statement. Experiment with two-dimensional and three-dimensional string arrays. Then go on to read about READ and DATA.

---

## READ and DATA—Another Way to Assign Values to Variables

---

So far you've learned that you can assign values to variables through direct assignment (`Color$ = "Yellow"`) and through INPUT statements (`INPUT "What color: "; Color$`).

There's also a third method that lets the computer do the assigning for you. The READ and DATA statements get information to assign to variables from within the body of the program itself. The keyword **DATA** is the first word in a DATA line, followed by a list of items (string literals and/or numeric constants) separated by commas:

```
DATA This is our finest hour, 3.14159
```

There can be many such **data lists** in the same program.

The keyword **READ** takes individual items from these data lists, beginning with items in the earliest list that it finds in a program, and assigns them to appropriate variables—that is, strings to string variables and numbers to numeric variables. For example:

```
READ Quote$  
READ Decimal
```

BASIC takes data items in order, one at a time, from the left of a list to the right. It remembers which items it has used. Once an item is used, BASIC moves on to the next one. When one whole data list is used up, BASIC moves on to the next data list.



### Do This

Use this short program to see how READ\DATA works. Type in and execute the following program:

```
FOR Name = 1 TO 7
  BTNWAIT
  READ Name$
  PRINT "The name I just found is "; Name$
NEXT Name
PRINT "That's all the names for now."
END PROGRAM

DATA Jason, Mary, Sarah, Sam, Pedro
DATA Throckmorton, Arthur
```

Each time you push the button on the mouse, BASIC will get another name from the data list and assign it to Name\$.

### The Data Pointer: Keeping Track of Used Items

When BASIC uses an item from a data list, it places a **data pointer** just past the item, as shown in Figure 9-10. The next time BASIC comes across a READ, it looks for the pointer in the data list and takes the item immediately following it. Then it moves the pointer ahead one item to be ready for the next READ.

READ\DATA is very handy when you have a lot of data you want to store in a program but don't want to assign to variables all at once. This pair of keywords can save you a tremendous amount of typing, especially when you use it with arrays.

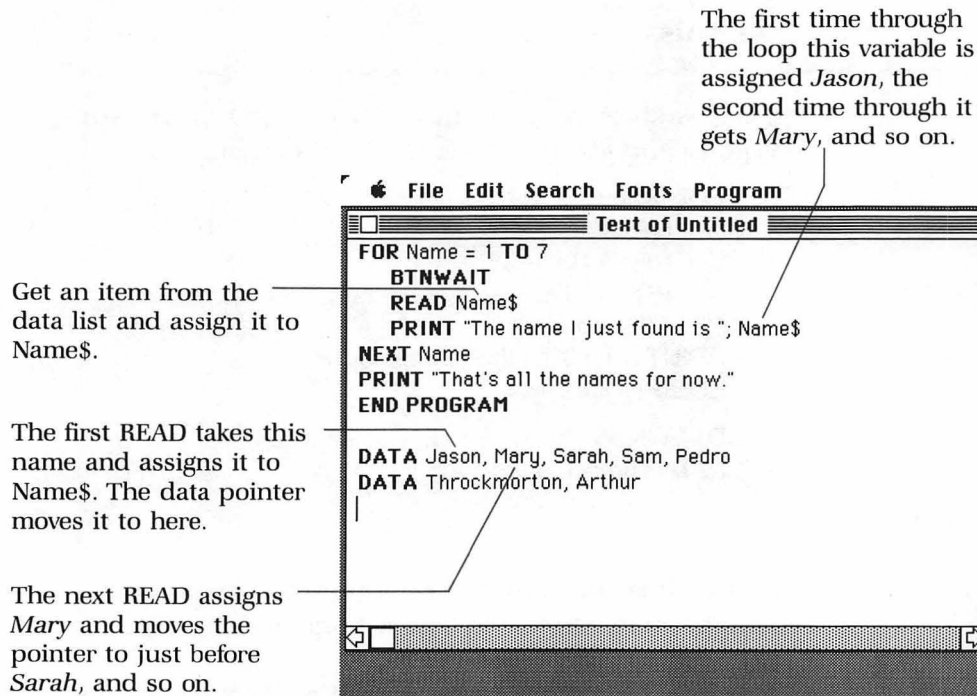


Figure 9-10 READ and DATA

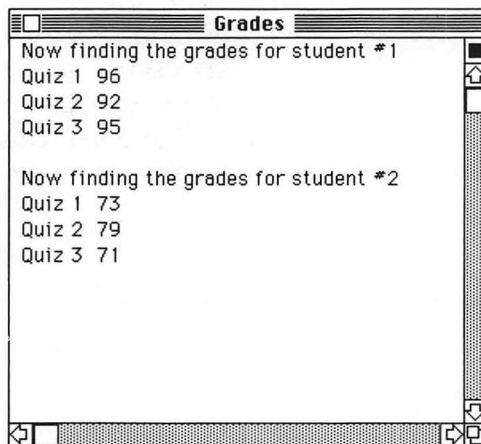


Figure 9-11 Output of students' grades DATA





## Do This

Go back to the *Grades* program you used earlier and substitute READ\DATA statements for INPUT, making appropriate changes to the PRINT statements. Your code should look like the program shown below. Now predict what the changes will do and execute the program.

```

Student.Count = 2
Quiz.Count = 3
DIM Grades(Student.Count, Quiz.Count)
FOR This.Student = 1 TO Student.Count
    BTNWAIT
    PRINT "Now finding the grades for student #"; This.Student

    FOR This.Quiz = 1 TO Quiz.Count
        PRINT "Quiz "; This.Quiz;
        READ Grades(This.Student, This.Quiz)
        PRINT " "; Grades(This.Student, This.Quiz)
    NEXT This.Quiz

    PRINT
NEXT This.Student

DATA 96, 92, 95      ! Student #1's scores
DATA 73, 79, 71     ! Student #2's scores
DATA                ! Reserved for a future class member

```

Figure 9-11 shows what your output window should look like.

Note that in this example the data lists come on two separate lines. You can have as many data lists as you want in a program. To add scores for new students, change the value of Student.Count to reflect the new total number of students and list the scores for each new student in a new data list. To increase the number of quizzes, change Quiz.Count and add the scores each student gets to the end of each student's data list (each list can be any length).



### For Your Information

**Including Commas in Data Items** The comma is the **delimiter** for data lists; you use it to let BASIC know where one data item ends and the next begins. But what if you want to include commas as part of a data item? The answer: enclose the data item in quotes. For example:

```
DATA This is here, That is there, "Here, there, and everywhere"  
DATA 100, 1000, "100,000"
```

Both data lists have three items in them. Be careful, though; the item "100,000" in the second list is a string, not a number; be sure to assign it to a string variable. Remember that data assignments, like any variable assignment, must match up all the way around: strings go into string variables, numbers go into numeric variables.

I'll leave it to you to figure out how to enclose quotes in a data item.



### Pop Quiz

#### Question 4

Add a segment to the *Grades* program that lets you ask for the score of any student on any quiz. The display for the new code must be something like this (user responses underlined):

Which student? 2

Which quiz? 1

The score is 73.



### Moving the Data Pointer: The RESTORE Statement

The data pointer moves along each time BASIC reads an item. If READ statements move the pointer beyond the end of all the lists, BASIC gives you an "Out of Data" error message. See for yourself:



#### Do This

Type in and run this program to make the computer gasp for data.

```
FOR Example = 1 TO 7
  READ Text$
  PRINT Text$;
NEXT Example
```

DATA This ,will ,blow ,up ,the ,building. ! Note the extra spaces

Figure 9-12 shows what happens.

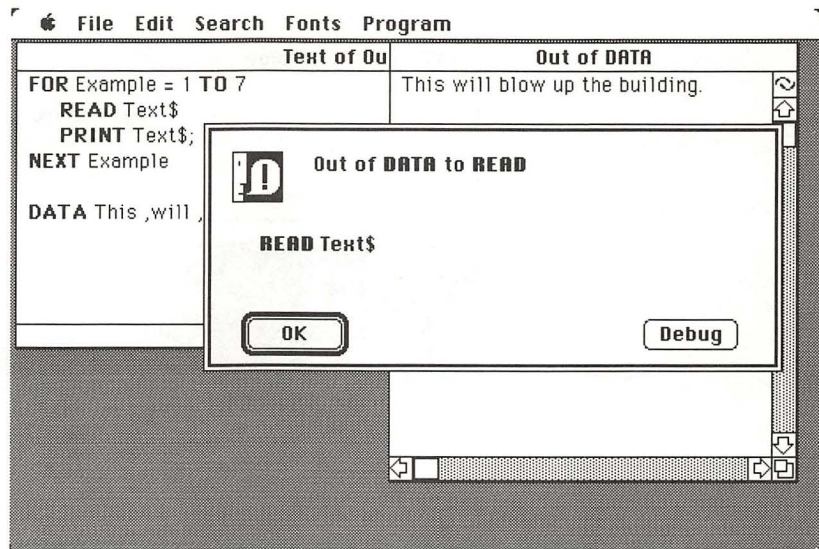


Figure 9-12 Gasping for DATA

If you want to reset the data pointer, you need to issue the keyword **RESTORE**. **RESTORE** sends the data pointer back to the first item in the program's earliest data list.

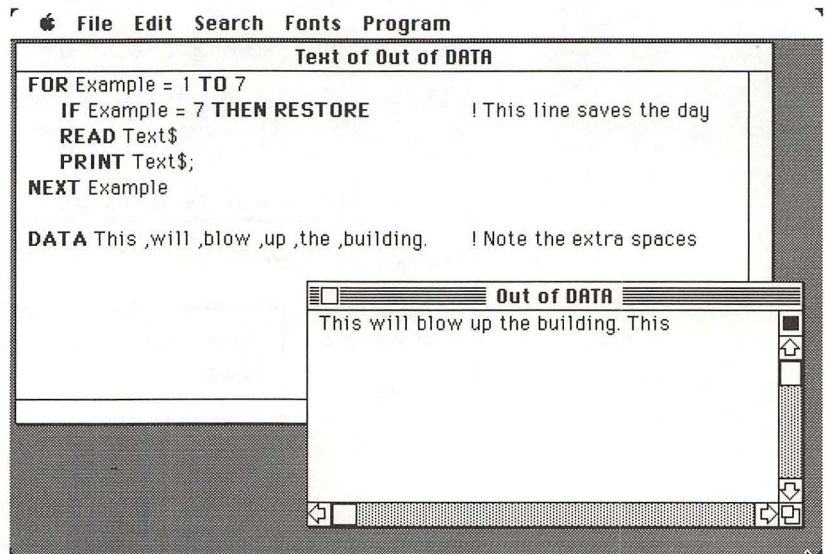


### Do This

Make this the second line in the previous program (just after **FOR Example...**) and then run the program.

**IF Example = 7 THEN RESTORE** ! This line saves the day

The complete listing and results are shown in Figure 9-13.



**Figure 9-13** Using **RESTORE** to re**READ** DATA



### RESTORE at a Specific Location

A slightly more complex but often more useful way to use RESTORE is to reset the data pointer to the beginning of a particular data list. Follow the keyword RESTORE with a label. When BASIC sees RESTORE *label*: it searches the program for a data list preceded by the same label, much as with the GOSUB statement. After it finds the label, BASIC moves the data pointer to just before the first data item in that list. The next time BASIC comes across a READ statement, it will begin picking up data items from the new pointer position.



#### Do This

Type in this program to see how RESTORE works with labels. (You can skip typing the comments.)

```

FOR Get.Em.All = 1 TO 25      ! This loops 25 times
  READ Use.It.Up              ! This picks up a new number for each pass
NEXT Get.Em.All               ! Increments the loop
                              ! They're all used up now—wasted!

RESTORE Fours:                ! Move the pointer to the first item
                              !   in the Fours: data list

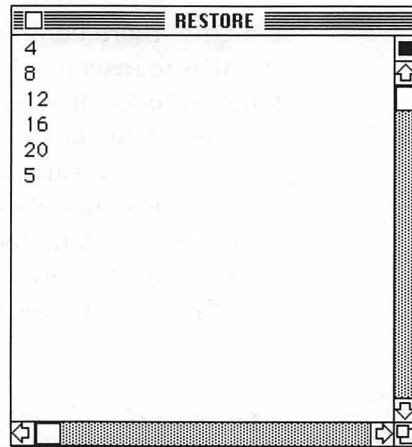
FOR Restoration = 1 TO 6
  READ Alive.Again            ! All five items in Fours: plus
  PRINT Alive.Again            !   one in Fives: get used again
NEXT Restoration              !   and printed again.

Ones: DATA 1, 2, 3, 4, 5
Twos: DATA 2, 4, 6, 8, 10
Threes: DATA 3, 6, 9, 12, 15
Fours: DATA 4, 8, 12, 16, 20
Fives: DATA 5, 10, 15, 20, 25 ! These are also available again
                                !   after RESTORE

```

Figure 9-14 shows your output.

Experiment with this program for a while until you really understand how RESTORE works with labels. Change the label to one of the other labels and see what happens. Use strings instead of numbers (careful!).



**Figure 9-14** How RESTORE works with labels

### **The READ\DATA\RESTORE Rules**

Here's a list of the rules that apply to READ\DATA\RESTORE. The best way to understand any that seem cloudy to you is to violate them. Programming errors can't hurt you, especially if you're the only one who sees them. Make all the mistakes you can now, while you're still new at it and have lots of excuses for messing up!

- Data lists can appear anywhere in a program. They can follow each other sequentially, or they can be separated from each other by other kinds of program lines.
- Each list must be preceded by the keyword DATA. DATA may (but doesn't have to be) preceded by a label.
- Any number of items can appear in a data list, as long as each item is separated from its neighbors by a comma.
- Strings and numbers can appear in the same data list.
- Strings in data lists need not be enclosed in quotes, but you must use quotes if you want to include a comma in a data item.

- READ statements must follow the general rules of BASIC syntax; strings cannot be assigned to numeric variables. Each READ statement moves the data pointer forward one item.
- You must have a data item for each READ, but you don't have to use all available data items.
- You can reuse data items by issuing RESTORE. Using RESTORE with a label resets the data pointer to the beginning of the data list with that label.
- Each time you run a program, the data pointer is reset to just before the first item in the first data list.



### Pop Quiz

#### Question 5

Change this program (which you first worked on in Session 8) so that it shows only the first names of five people whose full names appear in Data lists.

```
Search = 0
INPUT "Please type your first and last name: "; Name$
DO
    Search = Search + 1
    IF MID$(Name$, Search, 1) = " " THEN EXIT
LOOP

First.Name$ = LEFT$(Name$, Search - 1)
PRINT "Welcome to MacBASIC, "; First.Name$; "
```

After that, make it display again the first name of the third person in the Data list. Hint: get rid of the INPUT line.

### Cliffhanger

There's a lot more I could say about this stuff, especially about arrays. Arrays are so versatile that they could have several sessions dedicated to them—but that's for another book (Volume 2, anyone?).

Play with arrays on your own for now. Mix them into programs by using `READ\DATA\RESTORE` statements. Figure out some way to add graphics to the programs you've written for this session. Then go on to Session 10 to read about more graphics to delight your mind and warm your heart.

## Summary

---

### New Terms

---

**Array** variable structure in which a group of variables (the elements) are referenced by the same name but by unique numeric subscripts.

**Data list** one or more string literals or numeric constants following the keyword `DATA` and separated from each other by commas.

**Data pointer** pointer, invisible to programmer and user but recognizable by BASIC, which is positioned before the next available data item in a data list. If no more data items are available, the pointer remains at the end of the final data item in the program.

**Delimiter** ASCII character used to separate items. For instance, commas are delimiters between items in data lists, spaces are delimiters between keywords and non-keywords in programming statements, and colons are generally delimiters between labels and subsequent programming statements.

**Element** individual unit of an array, referenced by its numeric subscript.

**Multidimensional array** array with two or more lists referenced by the same array name but with more than one numeric in its subscript. There must be one such numeric for each dimension; each numeric is separated from its neighbors by a comma.

**Numeric subscript** numeric constant, variable, or expression appearing in parentheses immediately following an array name.

**Reference** naming of or calling attention to a variable or label by a program statement.

**Variable structure** manner in which a computer language stores values for variables.

### Menu Items

---

**Calculator** desk accessory used just like a simple four-function calculator. Results of all calculations are available for copying to Clipboard and subsequent pasting to a BASIC program.

**Find** find and highlight (that is, select) first occurrence following insertion point of string named in What to Find dialog box.

**Replace** substitute specified string with a replacement string. Options are set through What to Find dialog box. Replacement in program text happens at first instance of specified string following the insertion point.

**Replace All** substitute previously specified string with previously specified replacement string throughout the program.



---

## Programming Statements

---

**DATA** keyword marking beginning of data list. The only word that can precede DATA on a programming line is a label.

**DIM var(numexpr {(,numexpr)})** set up an array called *var* with *numexpr* + 1 elements. An array can have additional dimensions up to the limits of memory; you must give the number of elements for each dimension.

**READ var** retrieve the data list item immediately following the current position of the data pointer and store in variable *var*; move the data pointer past the retrieved item, to just before the following data item if another one exists.

**RESTORE [label:]** position the data pointer at the start of the first data list in the program. If RESTORE is followed by a label, position the data pointer at the start of the data list preceded by the matching label.

---

## Pop Quiz Answers

---

### Question 1

```

DIM Quiz(10)
Sum = 0
PRINT "Gimme ten values, please."
FOR Element = 1 TO 10
    PRINT "Value for Element #"; Element;
    INPUT " "; Value
    Quiz(Element) = Value
    Sum = Sum + Value
NEXT Element
Quiz(0) = Sum
PRINT
PRINT "Here are the values you chose: "
PRINT
FOR Element = 1 TO 10
    PRINT "Value for Element # "; Element; ": "; Quiz(Element)
NEXT Element
PRINT "The sum of all the values is "; Quiz(0)

```

### Question 2

This program just looks complex. It sets up a single-dimension string array with 11 elements. In elements 1 through 10 it puts a random ASCII character whose code is somewhere between 33 and 123; in element 0 it keeps a string made up of the random characters from each of the other 10 elements (it builds the string as it goes along). Finally, it displays the contents of all 11 elements.

**Question 3**

```

Student.Count = 2
Quiz.Count = 3
DIM Grades(Student.Count, Quiz.Count)

FOR Pupil = 1 TO Student.Count
  PRINT "Please enter the Grades for Student #"; Pupil

  FOR Test = 1 TO Quiz.Count
    PRINT "Quiz #"; Test;
    INPUT " "; Grades(Pupil, Test)
    Grades(0, Test) = Grades(0, Test) + Grades(Pupil, Test)
    Grades(Pupil, 0) = Grades(Pupil, 0) + Grades(Pupil, Test)
  NEXT Test

  PRINT
NEXT Pupil

FOR Test = 1 TO Quiz.Count
  Grades(0, Test) = Grades(0, Test) / Student.Count
  PRINT "Class average for Quiz "; Test; ": "; Grades(0, Test)
NEXT Test

FOR Pupil = 1 TO Student.Count
  Grades(Pupil, 0) = Grades(Pupil, 0) / Quiz.Count
  PRINT " Quiz average for Student "; Pupil; ": "; Grades(Pupil, 0)
NEXT Pupil

```

**Question 4**

Add this to the end of the program:

```

INPUT "Which student? "; This.Student
INPUT "Which quiz? "; This.Quiz
PRINT "The score is "; Grades(This.Student, This.Quiz)

```

**Question 5**

```

FOR Names = 1 TO 5
  Search = 0
  READ Name$
  DO
    Search = Search + 1
    IF MID$(Name$, Search, 1) = " " THEN EXIT ! 1 space in quotes
  LOOP
  First.Name$ = LEFT$(Name$, Search - 1)
  PRINT "Welcome to MacBASIC, "; First.Name$; "."

```

NEXT Names  
 RESTORE Ancient:  
 READ Name\$  
 PRINT "And a second welcome to you, "; Name\$

DATA John Scribblemonger, William Shakespeare  
 Ancient: DATA Charlotte Bronte  
 DATA Henry James, Emily Dickinson

## Commands, Menu Items, Keywords You Know So Far

### Commands and Menu Items

Alarm Clock	Option Key
Backspace key	Paste
Calculator	Replace
Clear	Replace the Rest
Copy	Run
Cut	Save
Halt	Select All
Key Caps	Undo
New	What to Find
Open	⌘ Key

### Programming Characters

+	-	/	*	=	>
<	;	!	\$	&	

### Programming Statements

ASC	LEFTS
BTNWAIT	LEN
CHR\$	MID\$
CLEARWINDOW	MOUSEB
DATA	MOUSEH
DATE\$	MOUSEV
DIM	OVAL
DO\LOOP	PAINT
ELSE	PLOT
END	PRINT
EXIT	READ
FOR...TO...STEP\NEXT	RECT
FRAME	RESTORE
GOSUB	RETURN
IF...THEN	RIGHTS
INPUT	RND
INT	TIME\$
INVERT	

---

**Bughouse**

---

This piece of code is supposed to print the question "When do we do more graphics?". And so it will, once you find and fix the nine or so bugs in it.

```
DIM Bugs(5)
DO
    Count = Count + 1
    WRITE Bugs(Count)
    Sentence$ = Sentence$ + " " + Bugs(Count)
    IF Count = 6 THEN EXIT
LOOP
Sentence$ = Sentence$ & "?"
PRINT Sentence$

DATUM When, do, we; do, more, graphics
```



## SESSION



# Graphics Revisited

---

**T**his session gives you more experience with the Mac's incredible graphics capabilities. You'll learn how to display text in different type styles and sizes, and you'll discover how to use the Mac's hidden special characters—predesigned pictures that live, hidden away, on your BASIC disk. You'll learn how to paint shapes with any one of 38 different patterns and how to increase the size of the graphics pen. Finally, you'll get some experience using a special shape called `ROUNDRECT`, a rectangle with rounded corners.

### **PRINT and GPRINT**

---

Long ago you learned about `PRINT`, BASIC's "display text" statement. `PRINT` is great—as long as you don't want to use text and graphics on the same horizontal plane.

#### **PRINT Destroys Graphics**

`PRINT` wipes out all graphics on its display line. For a demonstration, type in and run the following little program. Then read about why the destruction took place.



## Do This

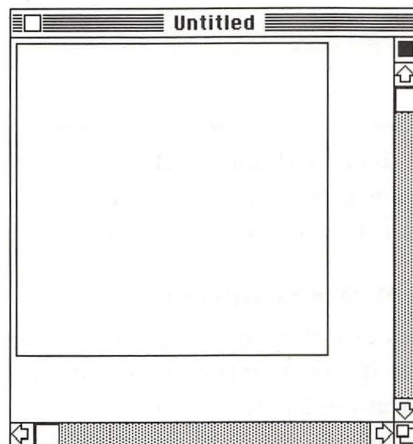
Type in and run this little program to see PRINT annihilate graphics.

```

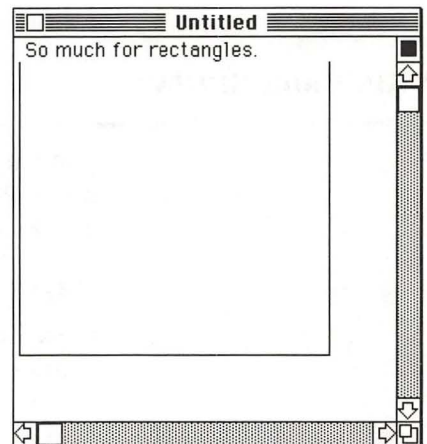
FRAME RECT 3, 3; 200, 200      ! Put up the graphics
FOR Stall = 1 TO 3000          ! Spin BASIC's wheels for a while
NEXT Stall                      ! to keep graphics alive
PRINT "So much for rectangles." ! Graphics whammo'd
  
```

Figure 10-1a and 10-1b give you a "before and after" view of the program's output.

The text insertion point, which determines where text appears, and the graphics **Pen point**, which decrees where graphics start, are separate entities. You moved the Pen to Column 3, Row 3 in your FRAME RECT statement, and the preset position for the text insertion point is the upper left corner of the window—more or less the same spot. So, even though you draw the rectangle first, the text appears on top of the first part of it and wipes out the top horizontal line.



**Figure 10-1a** Rectangle before execution of PRINT statement



**Figure 10-1b** Damage to graphics by PRINT statement

As it happens, there's a second kind of PRINT statement called GPRINT that lets you mix graphics and text. Not only that, but the text you use with your graphics can appear in a variety of sizes and styles.

### GPRINT: A Different Kind of PRINT

**GPRINT** means "Graphics PRINT." It's actually a kind of graphics statement. It gives you the freedom of graphics: you can put numbers and words anywhere you want without regard for the "usual" spacing between lines, you can change the typeface, and you can even make words overlap each other for special effects. But it makes you take responsibility for placing the text where you want it.



### Do This

Type this program into your computer exactly as it appears and then run it. (You can skip typing the comments.) Include the blank lines between sections. You'll see several new statements in the program, including SET PENPOS, GPRINT and SET FONT, all of which will ultimately be explained to your amazement and profound satisfaction.

PRINT "This looks like ordinary text."

Column = 0

First.Row = 28

Spacing = 16

SET PENPOS Column, First.Row

GPRINT "And this looks close to ordinary."

SET PENPOS Column, First.Row + Spacing

SET FONT 0

GPRINT "But this doesn't!"

! The very first column

! 28 dots down from top

! Dots in 1 "text" line

! Here's the next line's position

! Getting tricky here

! Here's the GPRINT font (typeface)

Figure 10-2 shows you what you should get. If you didn't get it, make sure you used GPRINT and not PRINT.

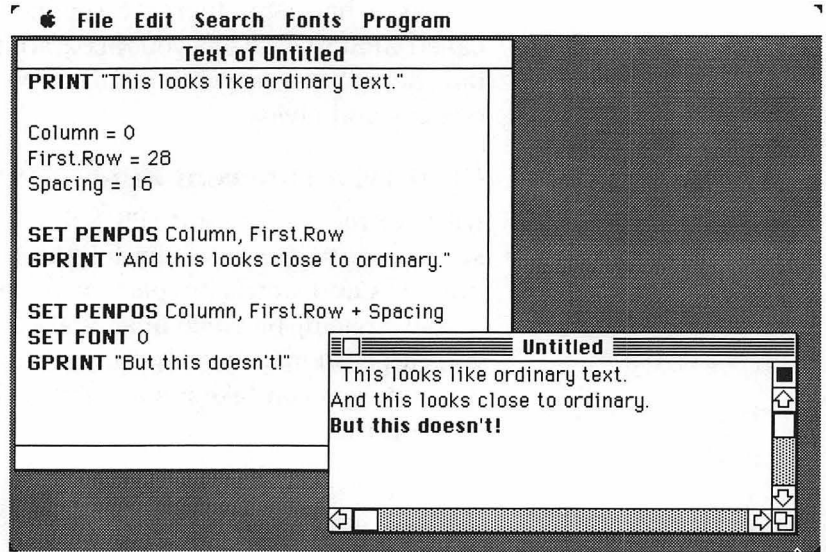


Figure 10-2 First output of GPRINT

### What the Program's About

The preceding program draws GPRINT text in the preset typeface, 12-point Geneva. Geneva is the type that BASIC ordinarily uses with PRINT statements. The "12-point" figure refers to the height of the characters. In printing there are 72 points to the inch; points on the Macintosh screen are off a bit, so 12-point type is a bit less than  $1\frac{1}{2}$ " high. A couple of sections down you'll learn how to change the size of GPRINT text. When you do that, the numbers will change; the proportions and the concepts will remain the same. Hang in there.

The first statement group in the program (just one line) simply displays ordinary text. But note what you've been taking for granted up to now: the PRINT statement knows in what column and row to start displaying the text, it knows what the type should look like, and it knows how big the type should be. I'll come back to all this.

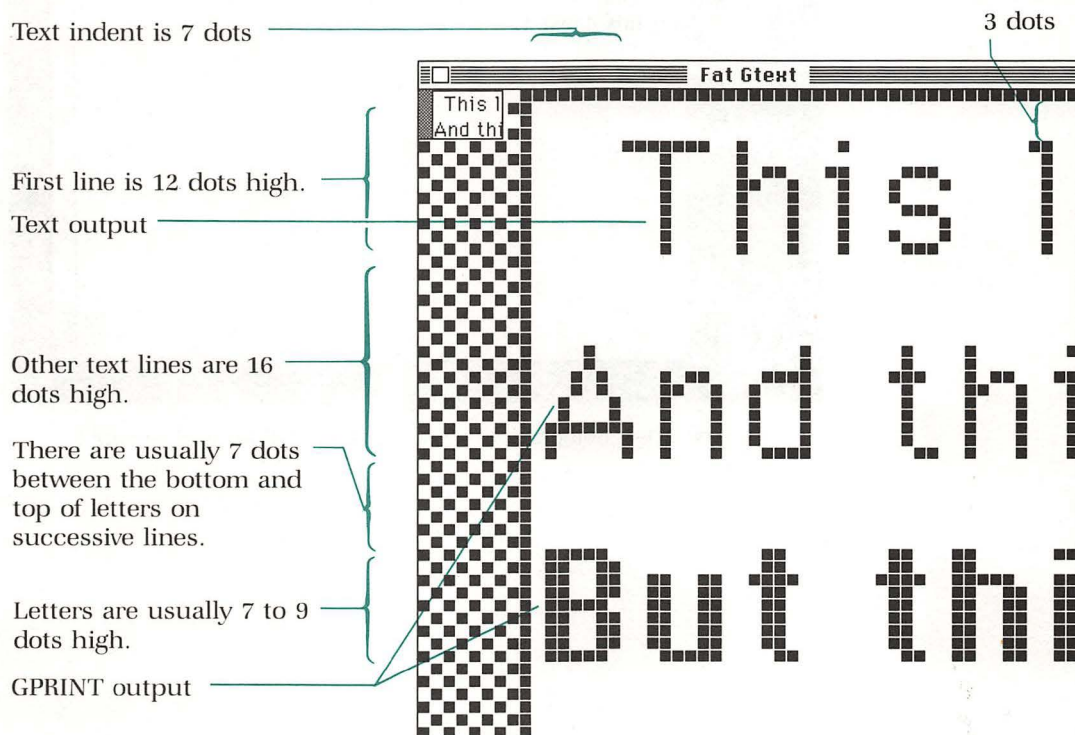
The second group of statements sets up some variables to be used in the rest of the program.



The third packet of lines moves the graphics Pen to the exact point on the screen where the new text is to appear—in this case, to column 0 (the extreme left edge of the window), row 28 (28 dots down from the top of the window, just below the title bar). **SET PENPOS** means “SET the PEN’s POSITION to the *Column* (,) *Row* you want.” BASIC draws graphics text from the new pen position; PENPOS is the position of the first dot in the *bottom left corner* of the character to be displayed. The syntax for SET PENPOS is exactly the same as for PLOT; but the action is different in that SET PENPOS positions the graphics Pen for action, while PLOT both positions the pen and takes action—it plots a point.

The final set of lines moves the Pen once more.

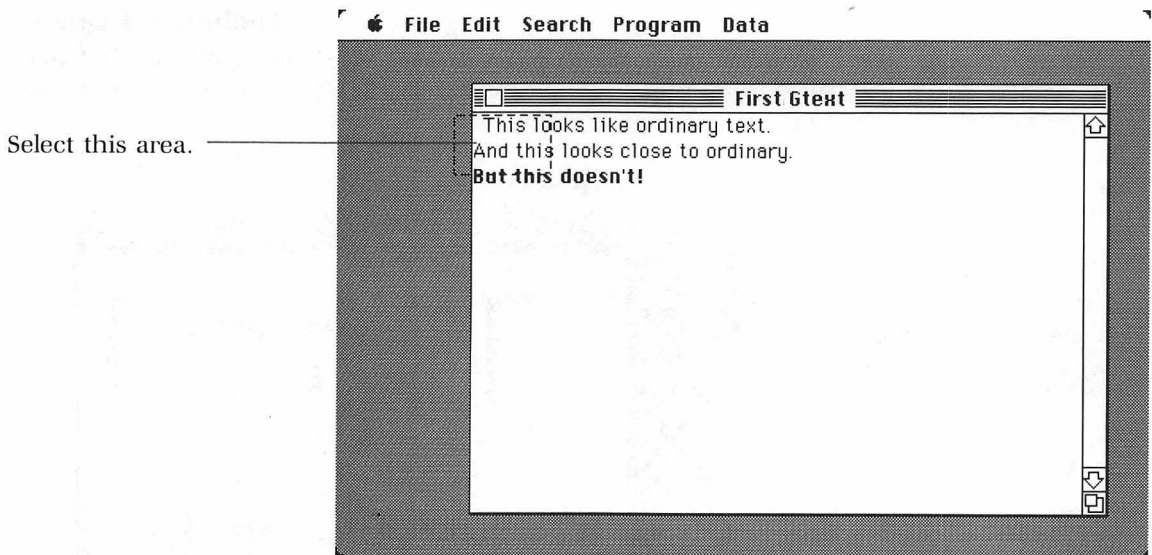
**Of Text Height, Dots, and Spacing: A Technical Explanation** Figure 10-3 lays out this entire section graphically. Check it out as you read the next few paragraphs to keep from getting lost.



**Figure 10-3** Close-up of text and GPRINT output

The expression *First.Row* + *Spacing* marks the row in which to display the first line of graphics text in this program. The variable called *Spacing* holds the value 16 because that's how high (in dots) an "ordinary" line of PRINT text is, including the usual space between lines. The text itself is 9 dots high, and the space between the lines is 7 dots.

*First.Row* holds the value 28—16 dots for an "ordinary" line height, plus 12 for the first line. The first line takes up less vertical space than do other lines; the space between the top line of "real" text and the top of the window is less than the space between two "ordinary" lines of text. Instead of 7 dots of space, there's only 3 (see Figure 10-3 again).



**Figure 10-4** Selecting area of text/GPRINT for a closer look



## Do This

**Use MacPaint To Help You** If you own a copy of MacPaint, Apple's graphics software for the Macintosh, you can use it to clear up any confusion you might feel about spacing between lines. Here's how:

1. Save the program you just worked with under the name *First Gtext*.
2. Create a MacPaint document out of the material on your screen now (assuming you've got both a listing window and output window) by using  $\text{⌘}$ -Shift-3.
3. Leave BASIC and start MacPaint.
4. Open the document called Screen 0, assuming you've not done this before on this disk. If you have, the new document will be the Screen document with the highest number listed.
5. Drag the selection rectangle around a small area between two lines, as shown in Figure 10-4.
6. Choose Fat Bits from the Goodies menu.
7. Count the dots between the lines.
8. Use the Hand to move the picture around so you can count the dots between the top line of text and the top of the window (see Figure 10-5).
9. Let loose an "Ah-HAH!" of understanding.
10. Leave MacPaint and get back into BASIC.
11. Open *First Gtext*.



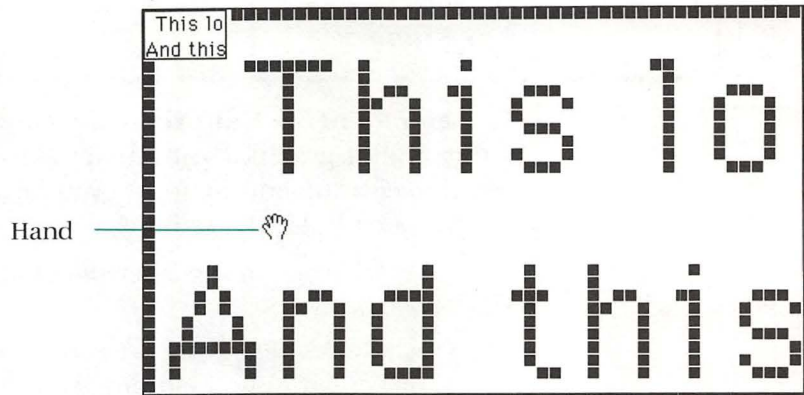


Figure 10-5 Using the Hand to move Fat Bits picture

## About Fonts

The second-to-last line of *First Gtext* changes the **font**. A font is a typeface or style of type. There are eleven different fonts available in BASIC.



### Do This

Type in and execute this program to display all the fonts available in Macintosh BASIC.

```

SET PENPOS 10, 12      ! Move Pen to proper position
FOR Style = 0 TO 12    ! Loop 13 times
  SET FONT Style        ! Get type style to use
  GPRINT "Here's a line in FONT "; Style
NEXT Style              ! Do the next round

```

The disk drive hums a lot when you run this program because the different fonts are stored on the disk; each font is retrieved when your program asks for it. The result of all this activity is shown in Figure 10-6.



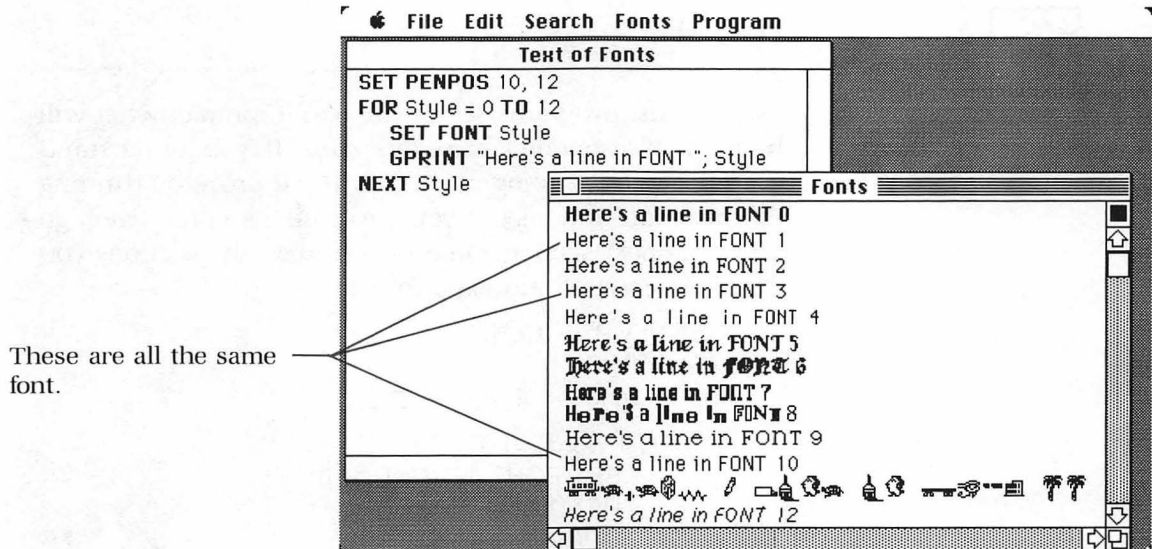


Figure 10-6 All of the fonts

You may have noticed that fonts 1, 3, and 10 are the same. Actually, font 1 is a reflection of something called the **application font**, the standard style for a particular application. The application in this case is the Macintosh BASIC language; the incredibly talented man who created this language, Donn Denman of Apple Computer, decided that the application font for BASIC would be font 3, Geneva. All regular PRINT text appears in font 3. If you don't change the font, GPRINT also does its stuff in font 3. There is no font 10 on the disk. When font 10 is specified, BASIC uses the application font.

Font 11, called Cairo, is made up entirely of little pictures and hieroglyphic characters. You can experiment with it on your own, using the CHR\$ function you learned in Session 8.

**Bizarre Play Time** Here's a strange little program to give you a feel for the assorted fonts from a decidedly different point of view.



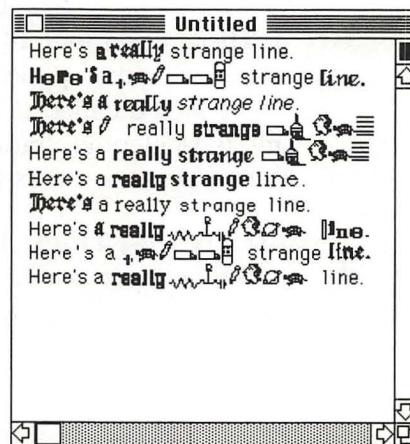
## Do This

Type in this program. Before you run it, predict what will happen. *Please don't skip this step.* If you understand everything that's going on (which you'll prove by running the program and seeing your predictions come true), go on to the next section. Otherwise, reread the sections you don't understand and experiment more.

```
SET PENPOS 10, 12
RANDOMIZE
FOR Strange = 1 TO 10
  FOR Word = 1 TO 5
    SET FONT INT(RND(13))
    READ Next.Word$
    GPRINT Next.Word$; " ";
  NEXT Word
  GPRINT    ! This starts a new line.
  RESTORE
NEXT Strange

DATA Here's,a,really,strange,line.
```

Figure 10-7 shows the really strange results.



**Figure 10-7** Really strange lines

While you're testing what you've learned so far, type in this little gem and see if you can understand what's going on.



### Do This

Type in this program. If you fail to analyze the code before you run it to check what you've learned so far, a very bad person will sneak into your computer room in the dead of night and pour molasses on your keyboard.

```
SET PENPOS 10, 18
Line$ = "Here's an even stranger line."
RANDOMIZE

FOR Strange = 1 TO LEN(Line$)
    SET FONT INT(RND(13))
    GPRINT MID$(Line$, Strange, 1);
NEXT Strange
```

You can see the output of these mysterious goings-on in Figure 10-8. Don't throw this program away; you'll need it for the next section. Save it in its current form if you're going to experiment with it (which of course you are encouraged to do).



**Figure 10-8** Even stranger line

## On Font Sizes

As you probably noticed, the output of the program you just wrote is hard to read; the type is kind of small. Luckily, Macintosh BASIC provides a way to make GPRINT characters larger than what's on your display right now. It uses the statement **SET FONTSIZE**.



### Do This

Modify the program you just wrote by adding this line right after the SET PENPOS statement:

```
SET FONTSIZE 18
```

Now run the program again.

**Type Comes in Many Sizes . . .** BASIC lets you use any font in any size you want. To repeat in greater detail what I said earlier, font sizes are in **points**. The term *point* is borrowed from printing; one point is  $\frac{1}{72}$ ". Thus something in 72-point type is one inch high, 36-point type is  $\frac{1}{2}$ ", 18-point type is  $\frac{1}{4}$ ", and so on. For technical reasons, the sizes don't exactly correspond to the sizes you see on the Macintosh screen, but they're close enough.

The preset size for MacBASIC type is 12 point, the preset font is font 3. Here's a program that takes the letter Q and shows it in a variety of sizes.





## Do This

Type in and run the program, which demonstrates various font sizes:

```
SET PENPOS 0, 100
FOR Character.Size = 1 TO 36
    SET FONTSIZE Character.Size
    GPRINT "Q";
NEXT Character.Size
```

Figure 10-9 shows the quirky, quizzical result.

**... But Some Sizes Are Better than Others** As you watch the Q's march toward you from off in the distance, you probably notice that some look better than others. It turns out that there's not just one set of characters called font 3 on the disk; there are seven sets of font 3. Each set is in a different font size—9, 10, 12, 14, 18, 20, and 24 points. If there's no font on the disk in the size that you ask for with SET FONTSIZE—say, 22 points—BASIC will design the character for you on the spot by scaling the image

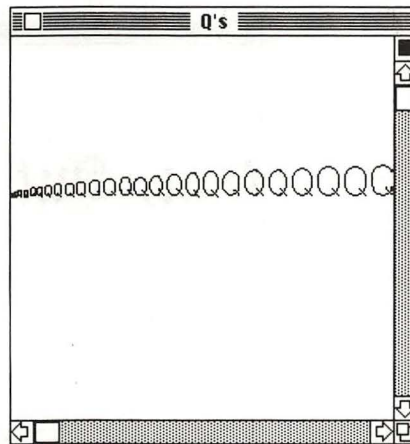


Figure 10-9 Q's

of a character that *is* available. A character that comes directly from a stored font looks better than one that doesn't, a **scaled character** whose size is an even multiple of an available one is almost as good, and a scaled character that BASIC has to fudge is—well—less good.

Here's an example of a font that's available only in one size—18 point—and that looks great in an even multiple of itself.



### Do This

Type in and execute the following program. It both demonstrates proper scaling and answers the Elizabethan question, "What do you say to a castle pet who has a little accident on the tapestry you're weaving?"

```
SET PENPOS 0, 100
SET FONTSIZE 36
SET FONT 6
GPRINT "Out, Out, Damned Spot!"
```




Figure 10-10 shows the damned output of the above.



**Figure 10-10** Historic quote

Table 10-1 shows you what fonts are available in what sizes. Keep in mind that anything smaller than 6 points is unreadable.

**Table 10-1** Available Fonts

Number	Name	Samples in Available Point Sizes
0	<b>System Font [Chicago]</b>	<b>12 point</b>
1	Application Font [Geneva]	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point
2	New York	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point, <b>36 point</b>
3	Geneva	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point
4	Monaco	9 point, 12 point
5	<b>Venice</b>	<b>14 point</b>
6	<b>London</b>	<b>18 point</b>
7	<b>Athens</b>	<b>18 point</b>
8	<b>San Francisco</b>	<b>18 point</b>
9	Toronto	9 point, 12 point, 14 point, 18 point, 24 point
 [11]	 [Cairo]	 [18 point]
12	<i>Los Angeles</i>	12 point, 24 point

## Special Font Characters

Back in Session 8 you learned about the CHR\$ function. You saw you could type an ASCII code number as CHR\$'s argument and get an ASCII character back. You also saw that CHR\$(217) produced a special character; in font 3, font size 12 it's a rabbit. Each separate font from 2 through 9 (and 12) has its own special character at CHR\$(217). In addition, font 0 has three special characters at ASCII codes 17, 18 and 20. Everything in font 11 is a special character. Fonts have different special characters for CHR\$(217), depending on the font size. Table 10-2 shows you all the special characters and how to generate them; Table 10-3 shows you all the characters in font 11.

**Table 10-2** Special Characters

CHR\$(217) produces these special characters in the fonts and sizes shown:


































































































	9 point	10 point	12 point	14 point	18 point	20 point	24 point
Font #2	♥	~	👤	🎵	♥	~	🤖
Font #3	🐸	📺	🐰	🦒	🐑	📺	🐱
Font #4	≡		🕯				
Font #5				🕸			
Font #6					🌸		
Font #7					🌸		
Font #8					🚗		
Font #9	♦		🐦	🍏	📦		🌿

Font #0 has three special characters in 12 point:

	CHR\$(17)	CHR\$(18)	CHR\$(20)
Font #0	⌘	✓	🍏



Table 10-3 Font 11

CHR\$(33) = 	CHR\$(57) = 	CHR\$(81) = 	CHR\$(105) = 
CHR\$(34) = 	CHR\$(58) = 	CHR\$(82) = 	CHR\$(106) = 
CHR\$(35) = 	CHR\$(59) = 	CHR\$(83) = 	CHR\$(107) = 
CHR\$(36) = 	CHR\$(60) = 	CHR\$(84) = 	CHR\$(108) = 
CHR\$(37) = 	CHR\$(61) = 	CHR\$(85) = 	CHR\$(109) = 
CHR\$(38) = 	CHR\$(62) = 	CHR\$(86) = 	CHR\$(110) = 
CHR\$(39) = 	CHR\$(63) = 	CHR\$(87) = 	CHR\$(111) = 
CHR\$(40) = 	CHR\$(64) = 	CHR\$(88) = 	CHR\$(112) = 
CHR\$(41) = 	CHR\$(65) = 	CHR\$(89) = 	CHR\$(113) = 
CHR\$(42) = 	CHR\$(66) = 	CHR\$(90) = 	CHR\$(114) = 
CHR\$(43) = 	CHR\$(67) = 	CHR\$(91) = 	CHR\$(115) = 
CHR\$(44) = 	CHR\$(68) = 	CHR\$(92) = 	CHR\$(116) = 
CHR\$(45) = 	CHR\$(69) = 	CHR\$(93) = 	CHR\$(117) = 
CHR\$(46) = 	CHR\$(70) = 	CHR\$(94) = 	CHR\$(118) = 
CHR\$(47) = 	CHR\$(71) = 	CHR\$(95) = 	CHR\$(119) = 
CHR\$(48) = 	CHR\$(72) = 	CHR\$(96) = 	CHR\$(120) = 
CHR\$(49) = 	CHR\$(73) = 	CHR\$(97) = 	CHR\$(121) = 
CHR\$(50) = 	CHR\$(74) = 	CHR\$(98) = 	CHR\$(122) = 
CHR\$(51) = 	CHR\$(75) = 	CHR\$(99) = 	CHR\$(123) = 
CHR\$(52) = 	CHR\$(76) = 	CHR\$(100) = 	CHR\$(124) = 
CHR\$(53) = 	CHR\$(77) = 	CHR\$(101) = 	CHR\$(125) = 
CHR\$(54) = 	CHR\$(78) = 	CHR\$(102) = 	CHR\$(126) = 
CHR\$(55) = 	CHR\$(79) = 	CHR\$(103) = 	CHR\$(127) = 
CHR\$(56) = 	CHR\$(80) = 	CHR\$(104) = 	CHR\$(128) = 
			CHR\$(129) = 

## Time to Experiment

Experiment for a while before you read any further. Mix various fonts and assorted sizes. Pepper your creations with special characters. Make a “Do Not Disturb” sign in Old English letters for your computer room door. Design a computer valentine for your mom (or somebody). Let your imagination go nuts. Then go on to learn about painting with patterns.

## MacBASIC Painting Patterns

In Session 6 you learned that the keyword **PAINT** filled a shape with a given pattern, but the only pattern you could use then was the one BASIC supplied automatically—BASIC black. There are actually 38 patterns available in BASIC; they're the same ones you'll find in the MacPaint program, which you probably own (if you don't, you should).

You select a pattern by using the keyword phrase **SET PATTERN** followed by the number of the pattern you want. You can use a numeric constant, variable, or expression to produce the pattern's number.



### Do This

Type in and run this little program which demonstrates how **SET PATTERN** works:

```
SET PATTERN 11  
PAINT OVAL 10, 10; 200, 200
```

The results appear in Figure 10-11.

As you can see, using **SET PATTERN** is absurdly simple. Just make sure to set the pattern you want before you paint a shape. The pattern that BASIC supplies if you don't say which one you want is pattern 0, solid black. Figure 10-12 shows all the patterns and their associated numbers (and what do you suppose pattern 19 is about?).

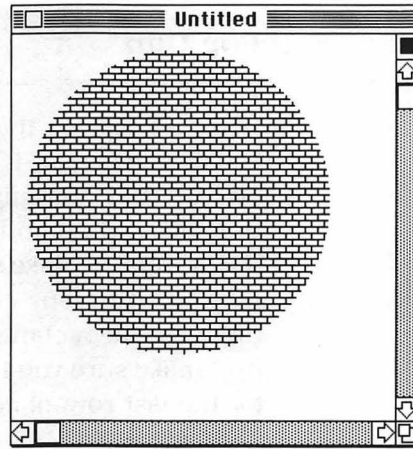


Figure 10-11 Pattern 11

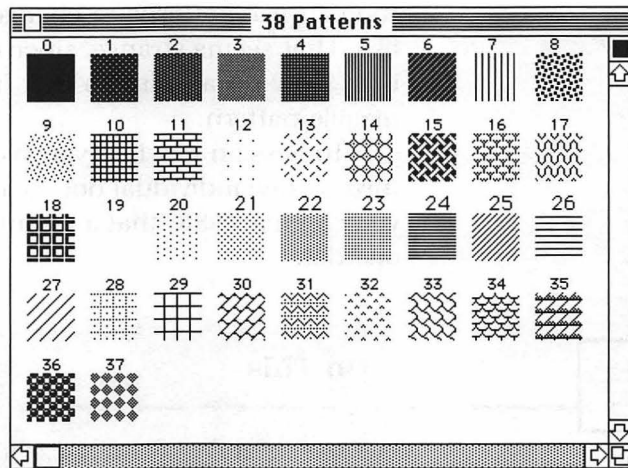


Figure 10-12 All 38 patterns



### Pop Quiz

Write a program that essentially duplicates Figure 10-12. You'll need to use SET PATTERN, SET PENPOS, PAINT RECT, and GPRINT. You might also want to use SET FONTSIZE (I did), SET FONT, and PAINT OVAL in place of PAINT RECT if you want to make yours fancier. Don't forget to use loops to condense your code; it's a waste of good memory to draw 38 little rectangles using at least four statements each! And make sure you have nine patterns in each row (except for the last row, of course).

### Thickening Points with PENSIZE

You can use patterns to change what each plotted point looks like. That seems strange, since each point is so small. The trick is to blow up a point so that it's big enough to show a recognizable pattern.

In Session 6, when you plotted a point, you told BASIC to turn on one individual dot. By using the phrase **SET PENSIZE**, you can tell BASIC that a "point" is to be made up of more than one dot.



### Do This

Type in and execute this little program to show the difference between standard points and points that have been set up via SET PENSIZE.

```
PLOT 100, 100  
  
FOR Stall = 1 TO 5000  
NEXT Stall  
  
SET PENSIZE 10, 20  
PLOT 100, 100
```



The standard size of a point is one dot high by one dot wide—in other words, a standard point is composed of one dot. It's as if BASIC assumes you want to SET PENSIZE 1, 1. When you SET PENSIZE 10, 20 you tell BASIC to make a single “point” with a Pen that's 10 dots wide by 20 dots high. Try drawing a line with this big Pen now.



### Do This

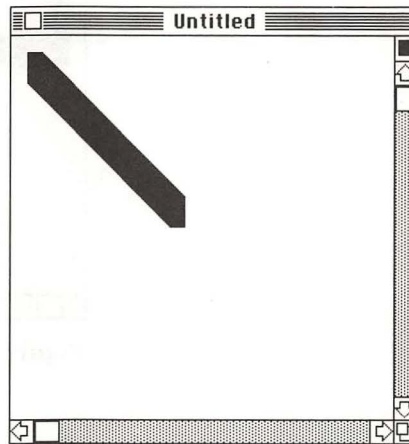
Change the last program so that PLOT draws a thick line.

```
PLOT 15, 15; 105, 105
```

```
FOR Stall = 1 TO 5000  
NEXT Stall
```

```
SET PENSIZE 10, 20  
PLOT 15, 15; 105, 105
```

Figure 10-13 shows the result.



**Figure 10-13** Thick line

Of course, what works with PLOT also works with the shapes. Here's an example that draws a rectangle. All you have to do is substitute FRAME RECT for the last example's PLOT.

```
FRAME RECT 10, 10; 100, 100
```

```
FOR Stall = 1 TO 5000
```

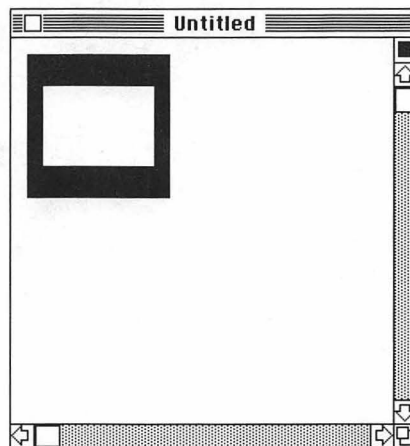
```
NEXT Stall
```

```
SET PENSIZE 10, 20
```

```
FRAME RECT 10, 10; 100, 100
```

You can really see the difference here between the height and width of each "point." The top and bottom are 20 dots wide, while the sides are 10 dots wide, as shown in Figure 10-14.

Change the program once more, this time substituting OVAL for RECT, and change the point thickness in the FRAME OVAL version to SET PENSIZE 1, 20, as shown below. That ought to get your creative juices flowing!



**Figure 10-14** Thick FRAME RECT



### Do This

Change the program to read like this:

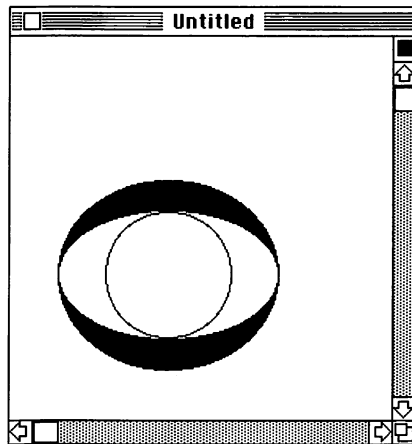
```
FRAME OVAL 60, 110; 140, 190
FOR Stall = 1 TO 5000
NEXT Stall

SET PENSIZE 1, 20
FRAME OVAL 30, 90; 170, 210
```

Two rather odd ovals are produced, as Figure 10-15 shows. Do you feel a sudden, uncontrollable urge to turn on your television set?

### On to Patterned Frames

Now that you know how to make thick points, you can start using patterns in your frames. Just set the thickness for the pen, name the pattern you want, and draw the frame. I'll give you one example; then you can go off on your own for a while.



**Figure 10-15** Thick and thin FRAME OVALs

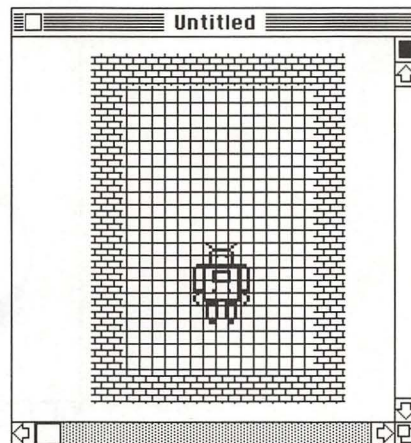


## Do This

Type in the first three lines of this example and execute them. Then add the rest of the code (you needn't type the comments). The whole example combines many of the elements you learned in this session so far; figure out what it does before you execute it.

SET PENSIZE 20, 20	! Use a thick, square Penpoint
SET PATTERN 11	! Choose the "brick" pattern
FRAME RECT 50, 10; 210, 230	! Outline a rectangular area
	! in the selected pattern
SET PATTERN 29	! Use a crosshatch pattern
PAINT RECT 70, 30; 190, 210	! Fill a rectangular area
	! with this new pattern
SET FONTSIZE 48	! Set up a very large character size
SET FONT 2	! Use a character from font 2
SET PENPOS 112, 170	! Position the Pen within the crosshatch
	! section of the drawing
GPRINT CHR\$(217)	! Display the character

The result of the program is captured in Figure 10-16.



**Figure 10-16** Robot in jail



Feel free to play with patterns for a while. Then come back to learn about a final graphics shape, ROUNDRECT.

## ROUNDRECT: A Rectangle with Rounded Corners

The **ROUNDRECT** shape bridges the gap between rectangles and ovals. Basically, a ROUNDRECT is a rectangle whose corners have been rounded off. As the programmer, you have control over the roundness. You set up a ROUNDRECT in much the same way you set up other shapes; you need to say how to draw the shape (FRAME, PAINT, or whatever) and give the upper left and lower right coordinates. In this case, you also need to do one thing more—give **roundness factors** for both the horizontal and vertical planes.

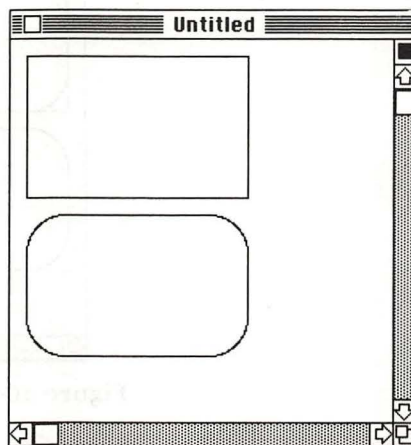


### Do This

Type in this short demonstration program and run it.

```
FRAME RECT 10, 10; 150, 100  
FRAME ROUNDRECT 10, 110; 150, 200 WITH 50, 50
```

Figure 10-17 shows you the result.



**Figure 10-17** RECT vs. ROUNDRECT

Both shapes are 90 dots high by 140 dots wide, but the curves on the second shape are extremely distinct. You can change the program to see the **ROUNDRECT** lying on top of a rectangle to make the distinction even more obvious. Just add a third line—but make it a **ROUNDRECT**.



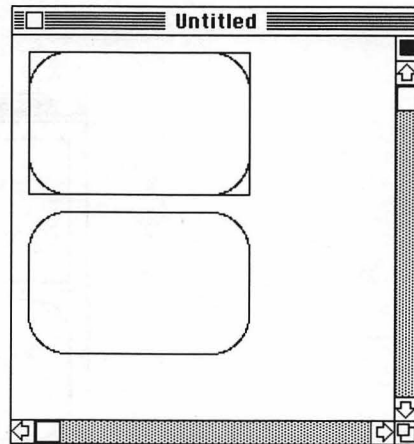
### Do This

Add a third line to the program, as shown below, and run it again:

```
FRAME RECT 10,10; 150, 100  
FRAME ROUNDRECT 10, 110; 150, 200 WITH 50, 50  
FRAME ROUNDRECT 10, 10; 150, 100 WITH 50, 50
```

Figure 10-18 shows the output.

The two numbers after the keyword **WITH** represent the horizontal and vertical roundness factors. The best way to see what they're about is—of course—by experimenting.



**Figure 10-18** ROUNDRECT within RECT



### Do This

Change the second line of the program by substituting 10 for the horizontal roundness factor; then run the program.

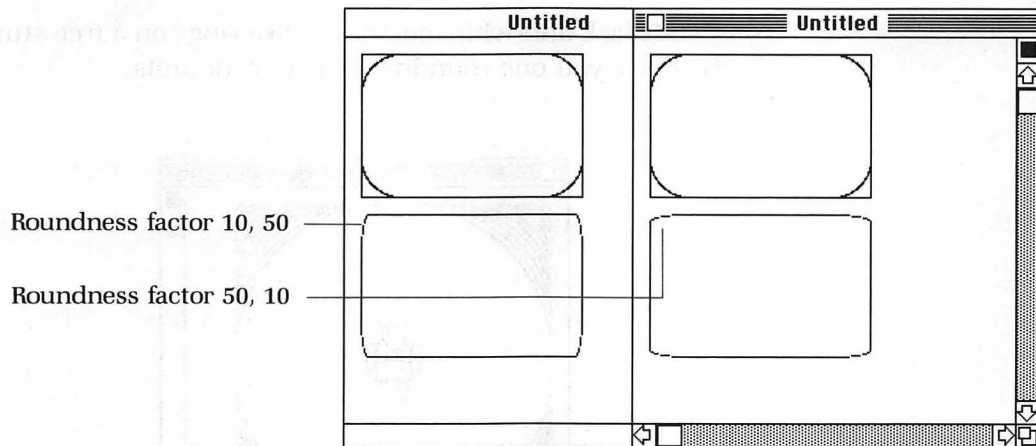
```
FRAME ROUNDRECT 10, 110; 150, 200 WITH 10, 50
```

Now change the horizontal factor back to 50, change the vertical factor to 10, and run the program again.

```
FRAME ROUNDRECT 10, 110; 150, 200 WITH 50, 10
```

See Figure 10-19 for the results.

Here's a program that lets you see what's happening with ROUNDRECT in a dynamic sort of way. Just type it in and run it. To see things happen, hold down the mouse button.



**Figure 10-19** Different roundness factors



## Do This

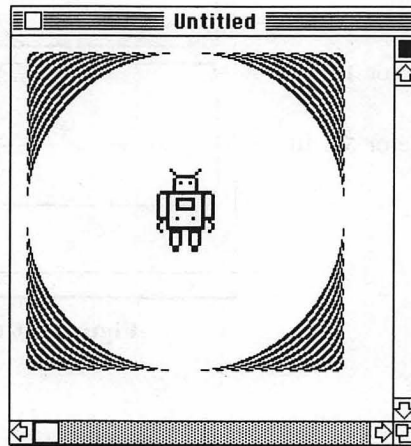
Type in and execute this program to demonstrate ROUNDRECT dynamically.

```
FOR Roundness = 10 TO 200 STEP 10
  INVERT ROUNDRECT 10, 10; 210, 210 WITH Roundness, Roundness
DO
  IF MOUSEB = 1 THEN EXIT
LOOP
NEXT Roundness

SET PENPOS 90, 125
SET FONT 2
SET FONTSIZE 48
GPRINT CHR$(217)
```

Figure 10-20 shows the final output.

The black-and-white curves are like rings on a tree stump; each shows you one roundness STEP of 10 units.



**Figure 10-20** Concentric INVERTed ROUNDRECTS with robot



See what you can do with `ROUNDRECT` on your own for a while. Try mixing text in the middle. Do a bunch of comparisons with ovals. Try using patterns with `PAINT ROUNDRECT`. Then go on to read the last section of this session to learn more about `SET`.

## ASK: The Other Side of SET

You used `SET` in several different ways in this session: to move the graphics Pen (`SET PENPOS`), to choose typefaces (`SET FONT`) and sizes (`SET FONTSIZE`), and to establish patterns (`SET PATTERN`) to be used with `PAINT` and with thick points and frames (`SET PENSIZE`).

All the keywords you've been using with `SET` are called **set-options**. Set-options are a special kind of system variable. They don't take arguments like regular numeric functions, but they are unlike other system functions in that you do have to give them parameters. You've already seen that you give a value to a set-option through `SET`. You can also find out what a particular set-option is set to, not in the usual way by saying something like `PRINT PATTERN` or `Number = FONT` (both of which make BASIC complain), but by using **ASK** to "ask" what the value is.

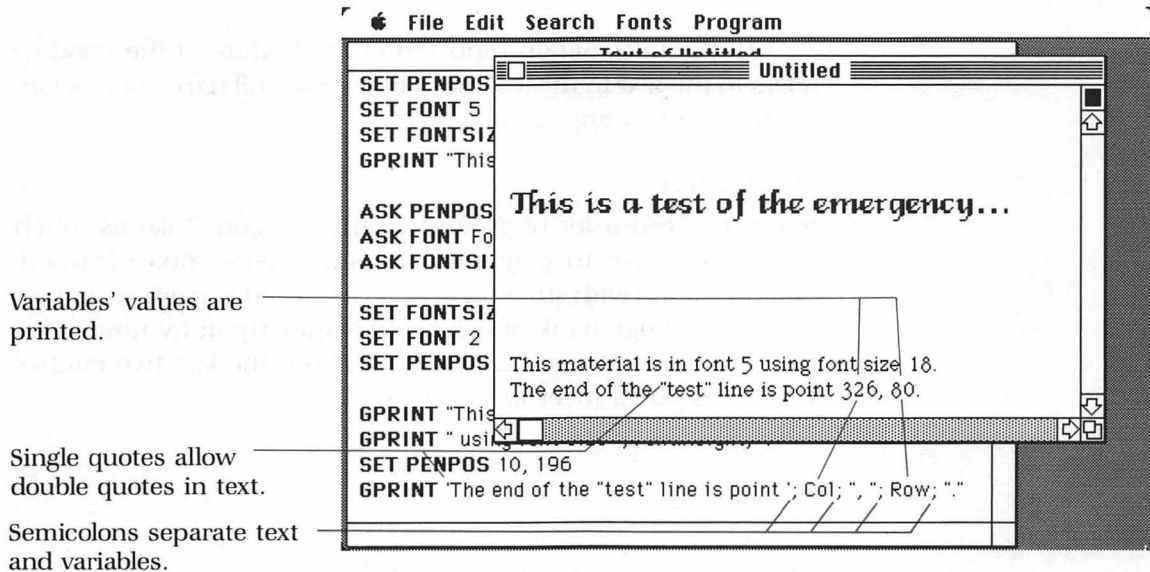


Figure 10-21 ASKING about SET options



### Do This

Type in and execute this program to see how ASK *set-option* works. Watch your punctuation carefully on the last line.

```
SET PENPOS 10, 80
SET FONT 5
SET FONTSIZE 18
GPRINT "This is a test of the emergency..."

ASK PENPOS Col, Row
ASK FONT Font.Num
ASK FONTSIZE Font.Height

SET FONTSIZE 12
SET FONT 2
SET PENPOS 10, 180

GPRINT "This material is in font "; Font.Num;
GPRINT " using font size "; Font.Height; ".";
SET PENPOS 10, 196
GPRINT "The end of the "test" line is point "; Col; ","; Row; "."
```

Figure 10-21 shows what you should get.

All the information reported at the bottom of the window refers to the text in the middle; all the ASK stuff happened before the bottom text appeared.

### Your Turn

You've covered a lot of ground in this session. Take as much time as you like to play with the statements you've learned. Combine them with statements you've learned in past sessions—you have a huge bank of material to draw upon by now. Then go on to the next session and learn about the last two control structures you'll meet in this book.

## Summary

---

### New Terms

---

**Application font** the preset typeface for any Macintosh application. BASIC uses font 3, size 12.

**Font** a particular style of type. BASIC has 11 different fonts, which can be specified through the use of *set-option* FONT.

**Font size** the height of a particular character in points. You choose the type size for a particular font using *set-option* FONTSIZE.

**Pen point** the tip of the graphics Pen. The size of the Pen point is determined by *set-option* PENSIZE.

**Point** any coordinate location on the Macintosh display; also, when referring to font size, about  $\frac{1}{72}$ " of type height.

### Programming Statements

---

**ASK *set-option numeric-variable*** assign to *numeric variable* the current value of the given *set-option*.

**FONT** a *set-option* in the range 0-12 determining the current font or type style. The preset font, called the application font, is font 1. In Macintosh BASIC, this is the same as font 3 (Geneva).

**FONTSIZE** a *set-option* in the range 1-127, with each increment representing 1 point of character height. Sizes below 6 are essentially unreadable. The preset size is 12.

**GPRINT** display the given character(s) in the predetermined FONT and FONTSIZE at the predetermined PENPOS.

**PATTERN** a *set-option* in the range 0-37 determining the pattern to be used with PAINT, or with PLOT and FRAME when the PENSIZE has been set sufficiently wide. The preset pattern is 0, all black.

**PENPOS** a *set-option* taking two parameters separated by a comma, with values between 0

**Scaled character** a character whose size is such that it doesn't exist in any font stored on the BASIC disk, thus making it necessary for BASIC to create it based on some size that *does* exist; so called because BASIC must scale an existing character to match the specified size. Also: called a lizard.

**Set-option** any of a number of specialized system variables affecting various system parameters. You give a parameter to a *set-option* through SET; you find out what the parameter is through ASK.

and 32767, representing the screen coordinate where the next graphics character will appear. The preset Pen position is 0, 0.

**PENSIZE** a *set-option* taking two parameters separated by a comma, with values between 0 and 32767, which represent the height and width (in dots) of the Pen point. The preset Pen size is 1, 1.

**ROUNDRECT** a graphics shape taking the ordinary parameters of a rectangle but ending in the syntactical phrase

WITH *horizontal.roundness, vertical.roundness*

in which the two variables are factors representing the degree of roundness of the shape's corners. (Some things just can't be adequately described in words.)

**SET *set-option numeric-expression*** assign to *set-option* the value of the given *numeric-expression*.

## Pop Quiz Answer

```

Left.col = 10
Top.Row = 10
Right.Col = 40
Bottom.Row = 40
DO
  SET FONTSIZE 9
  FOR Square = 1 TO 9
    SET PATTERN Pattern.Number
    PAINT RECT Left.Col, Top.Row; Right.Col, Bottom.Row
    SET PENPOS Left.Col + 10, Top.Row - 2
    GPRINT Pattern.Number
    Left.Col = Left.Col + 40
    Right.Col = Right.Col + 40
    Pattern.Number = Pattern.Number + 1
    IF Pattern.Number = 38 THEN EXIT
  NEXT Square
  IF Pattern.Number = 38 THEN EXIT
  Left.Col = 10
  Top.Row = Top.Row + 50
  Right.Col = 40
  Bottom.Row = Bottom.Row + 50
LOOP

```

## Commands, Menu Items, Keywords You Know So Far

### Commands and Menu Items

Alarm Clock	Option Key
Backspace Key	Paste
Calculator	Replace
Clear	Replace the Rest
Copy	Run
Cut	Save
Halt	Select All
Key Caps	Undo
New	What.to Find
Open	⌘ Key

### Programming Characters

+	-	/	*	=	>
<	;	!			
\$	&				

### Programming Statements

ASC	GPRINT
ASK	IF..THEN
BTWAIT	INPUT
CHR\$	INT
CLEARWINDOW	INVERT
DATA	LEFT\$
DATE\$	LEN
DIM	MID\$
DO\LOOP	MOUSEB
ELSE	MOUSEH
END	MOUSEV
EXIT	OVAL
FONT	PAINT
FONTSIZE	PATTERN
FOR...TO...STEP\NEXT	PENPOS
FRAME	PENSIZ
GOSUB	PLOT



PRINT	RIGHT\$
READ	RND
RECT	ROUNDRECT
RESTORE	SET
RETURN	TIME\$

---

## Bughouse

---

This wonderful program is supposed to draw five boxes, one beneath the other, with their numbers in them. All the numerics are OK, but there are bugs in the keywords and symbols.

```
SET PATTERN 5
SET PIXELSIZE 5, 5
SET FONTHEIGHT 18
SET FONT 6

Left = 10
Top = 10
Right = 60
Bottom = 35

FOR Box = 1 TO 5
  FRAME RECT Left, Top; Right, Bottom
  SET PENPOS Left + 8, Top + 20
  PRINT "Box"
  GOSUB Update:
NEXT BOX

Update
  Top = Top + 35
  Bottom = Bottom + 35

RETURN
```

# SESSION

## 11

# Advanced Decision Making

---

**T**his session deals with complex decision making in BASIC. In Session 4 you learned about IF...THEN...ELSE, BASIC's primary method of making decisions. If the outcome of an IF...THEN comparison is true, BASIC executes one statement; if the outcome is false, BASIC executes a different statement. BASIC is thus limited to executing just one of two statements per IF...THEN...ELSE construct.

BASIC has two other decision-making structures that aren't so limited: multiline IF...THEN\ELSE\ENDIF and SELECT\CASE\END SELECT. Multiline IF...THEN works like the IF...THEN...ELSE you already know, except that BASIC can execute as many statements as you want for either a true or a false outcome. SELECT CASE lets BASIC execute any number of statements depending on the value of some variable. Unlike IF statements, which have only two code pathways—one for true situations, one for false situations—SELECT CASE can have dozens of pathways.

BASIC makes all its decisions by making comparisons. If some condition is true, it does one thing; if the condition is false, it does another. (This applies even to SELECT CASE, in which the value of a variable is compared with FIXED values.) This kind of true/false decision making is based on boolean values, the third major concept that this session covers.

## Multiline IF...THEN\ELSE\ENDIF

This construct is very simple; you already know almost everything about it. Here's an example showing what it looks like.



### Do This

Type in this program to see how **multiline IF** works. Be sure to predict what will happen before you run it; you'll understand it right away.

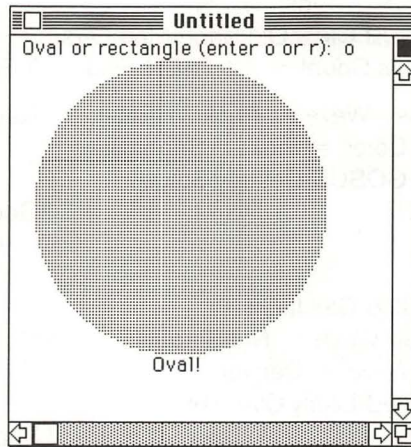
```

INPUT "Oval or rectangle (enter o or r): "; Answer$
IF Answer$ = "o" THEN
    SET PATTERN 23
    PAINT OVAL 15, 15; 200, 200
    SET PENPOS 90, 210
    GPRINT "Oval"
ELSE SET PATTERN 16
    PAINT RECT 20, 20; 180, 180
    SET PENPOS 70, 190
    GPRINT "Rectangle!"
ENDIF

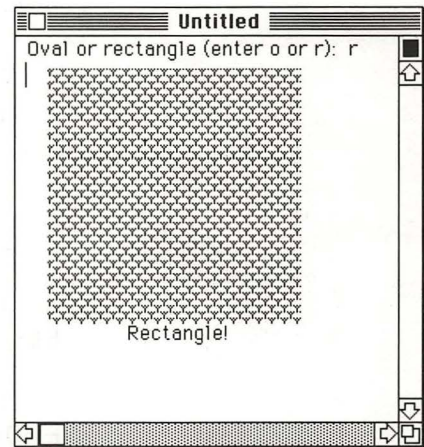
```

! Press Return after typing "THEN"  
! This happens if the operator  
! enters an "o". Note  
! that BASIC executes  
! all these lines, not just one.  
! BASIC comes here if the op-  
! erator enters anything except  
! an "o". Like single-line  
! IF, this ELSE part is optional.  
! ENDIF must appear at the end of  
! a multiline IF.

Figure 11-1 shows what happens.



**Figure 11-1a** Oval branch of Multiline IF...THEN\ELSE\ENDIF



**Figure 11-1b** Rectangle branch of Multiline IF...THEN\ELSE\ENDIF

Here are the essential things to remember about this construct:

- You can have BASIC execute any number of statements if a condition is true or if it's false.
- You must press Return after typing in the keyword THEN.
- The ELSE section is optional.
- You must end the construct with the keyword ENDIF.



## Pop Quiz

### Question 1

How would you write this same program using a single-line IF?

Here's a less simple example. You can't run this code because the subroutines it calls for don't exist, but you can see the possibilities:



```

IF Answer$ = "yes" THEN
  GOSUB Clear.Old.Band.Members:
  IF This.Count = 12 THEN Flag = 0 ELSE GOSUB End.It:

  IF New.Wave = Punk THEN      ! Nested multiline IF block
    Color = Pink + Bright.Blue
    GOSUB Destroy.Stage:
  ENDIF                        ! Goes with most recent IF
                                ! (didn't use optional
                                ! ELSE section)

  GOSUB Call.It.Off:
ELSE New.Wave = Not.Sinatra    ! Goes with outer IF block
  Old.Wave = Denver
  GOSUB Likely.Over.Thirty:
ENDIF                          ! Ends the whole block

```

There's nothing really complex about multiline IF; I'll leave you to experiment with it on your own. When you're done, go on to read about the more flexible and complex `SELECT\CASE\END SELECT` construct.

## SELECT—The Complex Decision Maker

---

Like IF, **SELECT** lets BASIC decide what statements to execute based on the evaluation of some expression. Unlike IF, however, **SELECT** isn't limited to either-or situations. **SELECT** runs blocks of code depending on the value of some variable. If that variable is in the range, say, of 1 to 100, **SELECT** can send BASIC in 101 possible directions.

All **SELECT** constructs begin with the keyword phrase `SELECT variable` and end with the phrase `END SELECT`. This first example shows **SELECT** in its simplest form.



### Do This

Type in this example of a simple SELECT construct. Predict what happens when you run the code.

```
INPUT "Type in a letter--A, B, or C: "; Letter$
SELECT Letter$
  CASE "A"
    PRINT "Ah, uppercase A."
    PRINT "My favorite letter."
  CASE "B"
    PRINT "The second letter of the alphabet."
    PRINT "I've always admired it."
  CASE "C"
    PRINT "I've never liked C's much."
  CASE ELSE
    PRINT "That's not an A, B, or C."
END SELECT
```

BASIC sees the variable name Letter\$ after the keyword SELECT and tries to find a **CASE** to match Letter\$'s value. Each CASE, followed by some literal value (it can't be a variable), marks the beginning of a block of statements. When BASIC finds a CASE value matching the contents of SELECT's variable, all the statements in that CASE's block (the block ends just before the next CASE or the END SELECT statement) are executed; if BASIC can't find any matching CASE, it executes the statements in the optional CASE ELSE block. When it finishes executing the block, BASIC jumps out of the construct and starts executing code at the next statement after END SELECT.

BASIC executes one and only one CASE block for each SELECT construct; if several CASE situations match, BASIC uses the first one it comes to. There *must* be at least one matching block.



## Do This

Use the comment marker character to isolate the CASE ELSE block from the program example you just entered. When you run the program, purposely type in a letter other than A, B, or C. Figure 11-2 shows the error message that you get.

While the CASE ELSE block is optional, you'd better use one if there's any possibility that BASIC won't find a matching CASE for the SELECT variable.

## SELECT with Ranges

You can use the keyword phrase *literal TO literal* on a CASE line to specify a range of values.

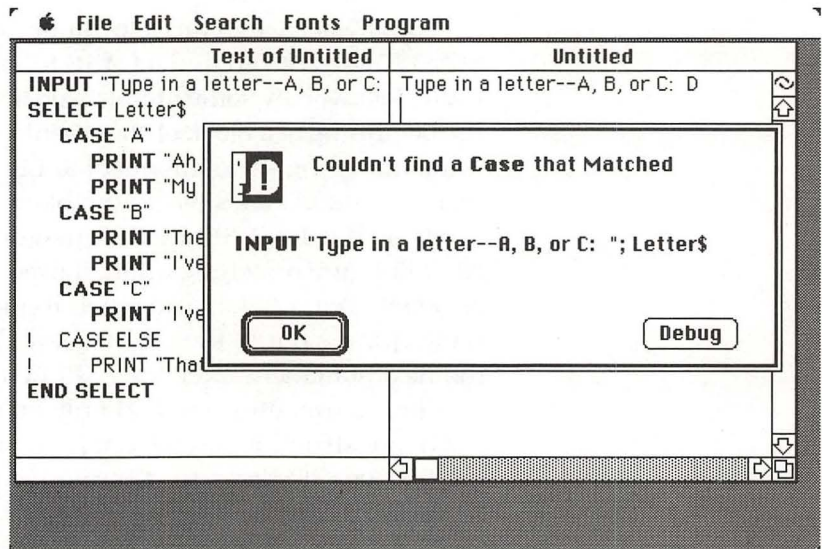


Figure 11-2 "Couldn't find a Case that Matched" error statement





### Do This

Remove the isolating comment characters from the CASE ELSE block. Then insert this new CASE situation into the example program between the CASE "C" block and the CASE ELSE block. Predict what will happen when you give BASIC a lowercase "x" for your answer; then run the program.

```
CASE "a" TO "z"
    PRINT "Sorry--I need a capital letter."
```

Lowercase "x" is within the range "a" TO "z", so BASIC executes the new block's code.



### Pop Quiz

#### Question 2

If you use it at all, why must the CASE ELSE block be the last one in the SELECT construct?

BASIC also lets you use the relational operators (<, >, <>) with CASE so you can specify ranges like > "z" and < "0".



### Do This

Insert this new CASE block before the CASE ELSE block. Then, as usual, predict the results and run the program. Don't let the comma throw you.

```
CASE < "A", > "Z"      ! What does the comma do?
    PRINT "No, it has to be an uppercase letter."
```



As you may remember from Session 8, BASIC compares strings based on the ASCII code values of the string characters. Therefore, BASIC executes this new block if the operator types any character with an ASCII value less than that of "A" (ASCII 65) or greater than that of "Z" (ASCII 90), which effectively takes care of all numbers or special symbols. If your code hadn't already provided for lowercase letters, this code would have.



### Pop Quiz

#### Question 3

Why is the CASE ELSE block redundant in the most recent version of this program?

**About That Comma: Lists of Parameters** You can use all the different parameters in combination as long as you separate them with commas. This next example shows how.

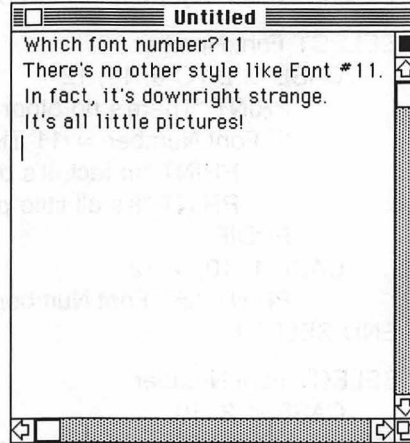


### Do This

Get a new listing window and type the following code. Then run the program which tells whether a font (whose number you supply) is unique.

```
INPUT "Which font number? "; Font.Number
SELECT Font.Number
  CASE 0, 2 TO 9, 11, 12
    PRINT "There's no other style like Font #"; Font.Number; "."
  IF Font.Number = 11 THEN
    PRINT "In fact, it's downright strange."
    PRINT "It's all little pictures!"
  ENDIF
  CASE 1, 10, > 12
    PRINT "#"; Font.Number; " is the same as Font #3."
END SELECT
```

Figure 11-3 shows one possible output. As you can tell from this example, you can include other multiline structures within a CASE block.



**Figure 11-3** Output of SELECT structure

### **Middle-of-Session Challenge**

Write a second SELECT structure for this program to fall hard on the heels of the one you just typed in. Using the value that Font.Number already has, make your new structure tell the user what font sizes are available for that particular font. Use the BASIC fonts chart (Table 10-1) to help you. While you're at it, make it show a sample of whatever font it's talking about. The size can be scaled, but you get 15 bonus points if BASIC uses one of the existing sizes. My solution's on the next page. You have one hour (set the Alarm Clock on your Mac).

**My Solution** I left the old **SELECT Font.Number** block intact and used it again, but I added a new first line to set a variable I use later.

```

Font.Size = 12                                ! The preset size (this is new code)
INPUT "Which font number? "; Font.Number

SELECT Font.Number                            ! Copied directly
CASE 0, 2 TO 9, 11, 12
    PRINT "There's no other style like Font #"; Font.Number; "."
    IF Font.Number = 11 THEN
        PRINT "In fact, it's downright strange."
        PRINT "It's all little pictures!"
    ENDIF
CASE 1, 10, > 12
    PRINT "#"; Font.Number; " is the same as Font #3."
END SELECT

SELECT Font.Number                            ! Rest of the code is new
CASE < 2, 10
    PRINT "The only size available is 12-point."
CASE 2, 3
    PRINT "This style has sizes 9, 10, 12, 14, 18, 20, and 24."
    IF Font.Number = 2 THEN PRINT "It's the only one with a size 36."
CASE 4
    PRINT "This type style comes in both 9- and 12-point."
CASE 5
    PRINT "This style comes only in 14-point."
    Font.Size = 14
CASE 6 TO 8, 11
    PRINT "Size 18 only for this style."
    Font.Size = 18
CASE 9
    PRINT "You've got 9-, 12-, 14-, 18-, and 24-point fonts for this one."
CASE 12
    PRINT "This type style comes in both 12- and 24-point."
CASE ELSE
    PRINT "That number isn't standard."
    PRINT "I can't tell you anything more about it."
END SELECT

GOSUB SAMPLE:                                ! Putting this here saves having to put
                                                ! it inside every CASE block.

END PROGRAM

```

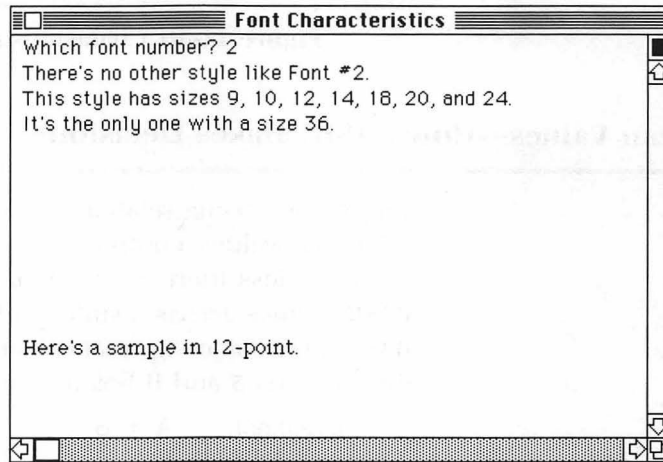
Sample:

```
SET FONT Font.Number  
SET FONTSIZE Font.Size  
SET PENPOS 8, 200  
GPRINT "Here's a sample in "; Font.Size; "-point."  
RETURN
```

RETURN

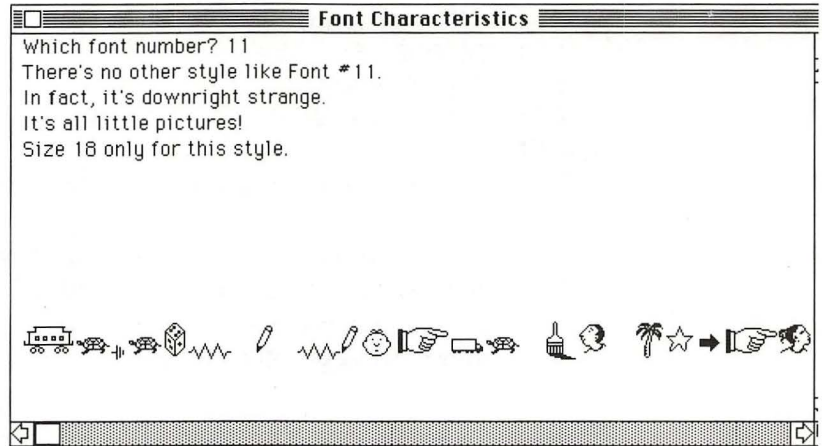
Figures 11-4a and 11-4b show some possible outputs of this program.

If you like, store this program under the name *Font Characteristics*. On your own time, put this whole program in a loop so that whoever's using the program has the option of getting information on other fonts.



**Figure 11-4a** Using SELECT to get font characteristics and sample





**Figure 11-4b** Characteristics and sample of Font 11

## Boolean Values—How BASIC Makes Decisions

You've been using relational values for most of this tutorial. All relational values compare two entities: something is greater than ( $>$ ), less than ( $<$ ), or equal to ( $=$ ) something else. When BASIC comes across a statement like `IF A = B THEN GOSUB C`: BASIC first comes up with a "true or false" decision. Assuming that A holds 5 and B holds 6, here's how BASIC thinks.

Statement:	<code>A = B</code>
Value for A:	5
Value for B:	6
Evaluation:	false
Decision:	Skip to next statement

Here's the same process, but now A holds 6 and B holds 6.

Statement:	<code>A = B</code>
Value for A:	6
Value for B:	6
Evaluation:	true
Decision:	<code>GOSUB C</code> :



### Do This

To see BASIC's thinking processes in action, type this minuscule program into a clean listing window and execute it.

```
A = 5
B = 6
PRINT A = B
```

The statement `PRINT A = B` tells BASIC to show you the result of a comparison between two entities, A and B. BASIC tells you directly that A does *not* equal B by giving the answer *false*. The `PRINT` statement says "Evaluate the statement 'A = B' and show the result."



### Do This

Change the program so that it looks like this:

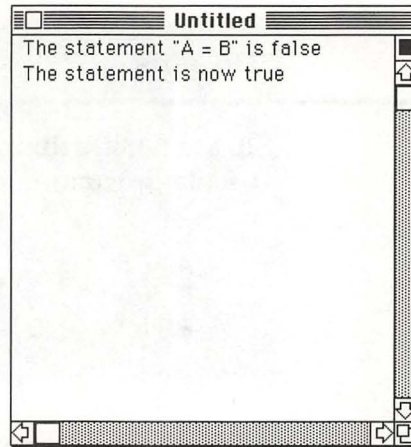
```
A = 5
B = 6
PRINT 'The statement "A = B" is ';
PRINT A = B
A = 6
PRINT "The statement is now ";
PRINT A = B
```

Figure 11-5 shows what you should get.



### For Your Information

Such "true or false" answers are called **boolean** results; similarly, the more formal name for a relational operator is a **boolean operator**. Both are named after the 19th-century mathematician George Boole.

**Figure 11-5** Booleans

### For Your Information

**Booleans in SELECT** BASIC does the same kind of boolean comparison to decide which CASE block to execute in a SELECT construct. When BASIC goes through a SELECT structure, it first gets a value for the SELECT variable—as in **SELECT Font.Number** from the program you worked on earlier. Then it goes to the first CASE line, gets the first value it finds there, and compares it to the value of the SELECT variable. If the comparison evaluates as true, BASIC executes that CASE block and then leaves the SELECT structure. If not, BASIC looks at the next value, does another comparison, and so on. Of course, you don't see these true-false results directly; just as with IF..THEN constructs, BASIC does the boolean evaluations “internally” and shows the results by executing proper blocks of code or by making appropriate branches.

### Some Hidden Boolean Operators

The chart shows you the boolean operators you know about, plus a few others you can get by using the Option key



on your Macintosh keyboard. For comparison purposes, suppose that A holds the value 5 and B holds 6.

Operator	Comparison	A to B
=	equals	false
<	less than	true
>	greater than	false
<> or ≠	not equal	true
>= or ≥	not less than	false
<= or ≤	not greater than	true

- To get  $\neq$  press Option-= (hold down the Option key while typing "=")
- To get  $\leq$  press Option-, (hold down the Option key while typing ",")
- To get  $\geq$  press Option-. (hold down the Option key while typing ".")

### The Boolean Data Type

When BASIC produces them as the result of a boolean comparison, the values *true* and *false* are neither numbers (as you may have suspected) nor strings (which may surprise you). They are in fact **boolean values**. Boolean constitutes a third data type in BASIC. Its symbol is  $\sim$  (pronounced "boolean"); its key is the one at the upper-left corner of the keyboard.



#### Do This

Type in yet another minuscule program to experience the newest BASIC data type.

```
Less = 5
More = 6
Truth~ = Less < More
PRINT Truth~
```

When you execute this program, BASIC will display "true". Do it now.



You assign values to **boolean variables** the same way you do to any variable:

```
This = true
```

```
That = false
```

Just as you can have numeric and string arrays, you can have boolean arrays. You DIM them the same as you would any array. For instance, to set up a two-dimensional boolean array called *Flags* with 11 elements in the first dimension and 6 in the second, you would type

```
DIM Flags(10, 5)
```

Of course, each element can have only one of two values—true or false:

```
Flags(5, 3) = true
```

```
Flags(6, 1) = This > That
```

The second example says “Set element 6, 1 of boolean array *Flags* to hold the evaluation of the statement ‘the value of *This* is greater than the value of *That*.’” Assuming *This* holds 3 and *That* holds 10, then *This* > *That* evaluates as false; BASIC will set *Flags*(6, 1) to false.



### For Your Information

**Undefined Booleans** Just as BASIC sets undefined string variables to hold the null string and undefined numeric variables to hold 0, it sets undefined booleans to hold the value *false*:

```
PRINT Boole ! yields false since Boole is undefined
```

## Booleans as Flags

A **flag** is a variable you use to signal particular events or changes. In Session 7's Dice Game program I used two flags, *Winner* and *Endit*; if *Winner* was set to 1, it meant that a player had made his or her "point," and if *Endit* held a 1, it meant that a player had decided to end the game. Booleans make perfect flags because they can hold only the values *true* or *false*, and a flag is either set (true) or it is *not* set (false). Here's an example of how to use a boolean as a flag:

```
IF Answer$ = "y" THEN On~ = true
```

```
...  
IF On~ THEN GOSUB Light.It:
```

That's not a typo; you don't have to say `IF On~ = true`. In fact, you can't; if you try you'll get a "Type mismatch error" message.

"But what do I do if I want the program to do something when a boolean holds the value *false*?" you may wonder. You use the keyword **NOT**, like this:

```
IF NOT On~ THEN GOSUB Turn.It.Off:
```

The form `IF << boolean variable >> THEN...` means "If the value of the boolean is *true*, then do something." By the same token, the form `IF NOT << boolean variable >> THEN...` means "If the value of the boolean is *false*, then do something."

## Keeping Boolean Values Straight

You can stop yourself from being confused if you remember that booleans have one-dimensional existences. All they can do is handle various permutations of true and false states. Here's an example showing the most confusing that booleans will ever get.



## Do This

Type in this final boolean sample program and run it:

```
Smaller = 5  
Larger = 6  
Judgment~ = Smaller > Larger  
PRINT Judgment~  
PRINT NOT Judgment~  
PRINT NOT NOT Judgment~
```

Figure 11-6 shows what you get.

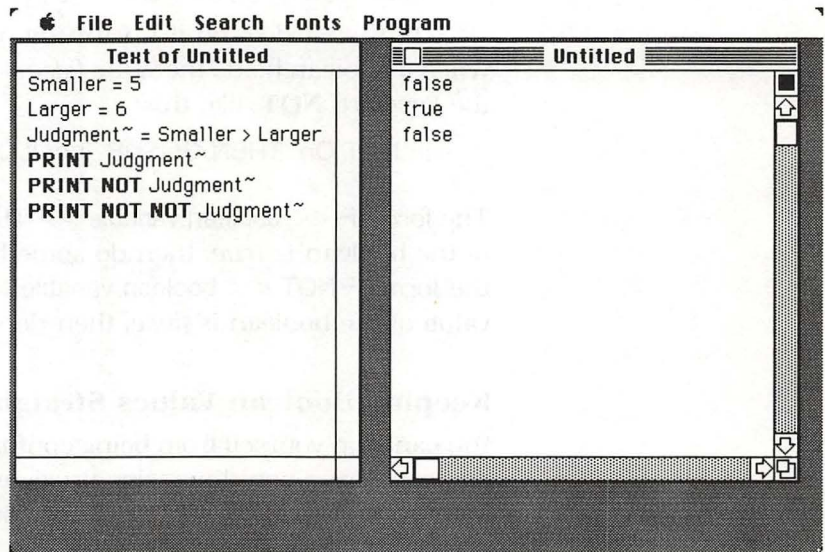


Figure 11-6 NOT and NOT NOT

BASIC gives you “false” and then “true” and then “false” again—which makes sense if you think about it. If you live in a black-white, on-off, yes-no world as booleans and computers do, something is either true or false. If something is not true, it is false; if something is not false, it is true. Since the value of *Judgment* is false, then NOT *Judgment* must be true. Finally, since NOT negates (that is, makes opposite) its object, to say something is not “not false” means that something is not “true”—which means false!

That's as bad as it gets; in fact, you'll seldom (if ever) see “NOT NOT”.

### **MOUSEB: A Boolean Keyword**

In an earlier session you learned about MOUSEB, the system function that returns the value 1 if somebody is holding down the mouse's button and 0 if not. There's an alternate form, **MOUSEB**, that also reflects the state of the mouse button, except that it returns a boolean value. If the mouse button is down, **MOUSEB** yields *true*; if the button is not down, **MOUSEB** returns *false*. This lets you say things like IF **MOUSEB** THEN GOSUB Do.Something: or IF NOT **MOUSEB** THEN Button\$ = “Up”. Very handy.

## **Time to Solidify Learning**

---

You'll be using flags, booleans, complex decision making, and almost everything else you've learned in this book in the next session. For now, experiment on your own. Write a program using IF...THEN\ELSE\ENDIF, SELECT, and booleans, both to get some practice using these new tools and to prepare yourself for the next and final session—in which you'll plan and write *The Great American Sheep Race*!



## Summary

### New Terms

**Boolean** referring to a true/false condition.

**Boolean variable** variable whose name ends with the boolean symbol `~`. Boolean variables can hold one of only two values—true or false.

**Flag** a variable holding one of two values (usually true/false, 1/0, nonzero/zero or positive number/negative number); BASIC uses it to determine which of two paths to follow.

### Programming Statements

`≠` relational (or boolean) operator meaning *not equal*.

`≥` relational (or boolean) operator meaning *not less than*.

`≤` relational (or boolean) operator meaning *not more than*.

`~` marker character for boolean data type (just as `$` is string marker).

**IF...THEN\ELSE\ENDIF** multiline IF...THEN ...ELSE construct.

**MOUSEB~** alternate, boolean form of the system function MOUSEB. This form allows you to use constructs such as IF MOUSEB~ THEN... and IF NOT MOUSEB~ THEN... instead of IF MOUSEB = 1 THEN... and IF MOUSEB = 0 THEN....

**NOT** negate a boolean value; that is, produce *false* if the boolean variable holds true, and vice versa.

**SELECT var\CASE literal\[CASE ELSE] END SELECT** If *var* holds a value matching a literal that appears after any occurrence of the keyword CASE, execute the statements in the CASE's code block. If there's no matching literal, execute the statements in the CASE ELSE block. A CASE's literal can be part of a range (*literal TO literal*), have a relational operator before it (for example, *> literal*), or take a list of literals, ranges, and/or relationals.

### Pop Quiz Answers

#### Question 1

```
INPUT "Oval or rectangle (enter o or r): "; Answer$
```

```
IF Answer$ = "o" THEN GOSUB Oval: ELSE GOSUB Rectangle:
END PROGRAM
```

```
Oval:
```

```
SET PATTERN 23
```

```
PAINT OVAL 15, 15; 200, 200
```

```
SET PENPOS 90, 210
```

```
GPRINT "Oval!"
```

```
RETURN
```

Rectangle:

```
SET PATTERN 16
PAINT RECT 20, 20; 180, 180
SET PENPOS 70, 190
GPRINT "Rectangle!"
```

RETURN

## Question 2

CASE ELSE must be the last block in the structure because BASIC carries out the code for the first valid CASE alternative it finds. By definition, the CASE ELSE block handles any and all alternatives that earlier CASE blocks missed. If CASE ELSE were first, for example, BASIC would execute its code immediately and ignore everything else.

## Question 3

CASE ELSE is redundant here because CASE < "A", > "Z" handles every possible situation that the other CASE blocks don't already handle. In English, this block says "Execute this code if what the user typed is not a capital letter."

## Commands, Menu Items, Keywords You Know So Far

### Commands and Menu Items

Alarm Clock	Option Key
Backspace key	Paste
Calculator	Replace
Clear	Replace the Rest
Copy	Run
Cut	Save
Halt	Select All
Key Caps	Undo
New	What to Find
Open	⌘ Key

### Programming Characters

+	-	/	*
=	>	<	;
!	\$	&	≠
≥	≤	~	

### Programming Statements

ASC	LEN
ASK	MID\$
BTNWAIT	MOUSEB
CHR\$	MOUSEB~
CLEARWINDOW	MOUSEH
DATA	MOUSEV
DATE\$	NOT
DIM	OVAL
DO\LOOP	PAINT
ELSE	PATTERN
END	PENPOS
EXIT	PENSIZ
FONT	PLOT
FONTSIZE	PRINT
FOR...TO...STEP\NEXT	READ
FRAME	RECT
GOSUB	RESTORE
GPRINT	RETURN
IF\ENDIF	RIGHT\$
IF...THEN	RND
INPUT	ROUNDRECT
INT	SELECT\END SELECT
INVERT	SET
LEFT\$	TIMES\$

**Bughouse**

---

Here's the last "directed" debugger in the book; after this, they all count! The following program has half a dozen bugs.

```
DO
  IF MOUSEB = "true" THEN CLEARWINDOW
  SET PENPOS RND(250), 0
  GET PENPOS Column, Row
  SELECT COLUMN
    IN CASE < 100
      SET PATTERN 23
      PAINT RECT 10, 10; 100, 200
    CASE 100 THROUGH 200
      SET PATTERN 16
      PAINT OVAL 100, 240, 245
    CASE > 200
      PATTERN 4
      PAINT ROUNDRECT 100, 200; 300,
        275 WITH 50, 50
  FINISH SELECT
ENDIF
LOOP
```

## SESSION

# 12

## Program Planning

**T**he major focus of this final session is the planning and writing of a complex program. The program you're going to write uses most of the statements you've learned in this tutorial, tying them together in one overall context. You'll see what elements go into the planning of a program that might seem large to you—about 150 lines of code. Actually, 150 lines is a relatively short program; but you can use the method you'll be introduced to here for all your programming. This method, which uses something called top-down programming and another technique called pseudocode development, certainly isn't the only one available for program planning, but it has the distinct virtue of working.

The program itself—*The Great American Sheep Race*—demonstrates some elementary principles of animation and introduces you to the ambulatory habits of that marvelous mammal, *Ramis Macintoshius*.



### **A Heartfelt Warning**

This session is different from the others; rather than immediately having you type in a program or program segment, I ask you this time to wait, to think, to plan, to think and plan more, and finally to plan even more before you even touch the computer. Every programmer eventually learns that careful planning pays off. Of course, planning takes time. That's where the problem starts: with a machine as inviting as the Macintosh, you're always tempted to hit the keyboard *right now*, as soon as an idea hits. "Get that code down—transfer that vision to the computer immediately! Strike while the Muse is hot!" This is folly.

Spontaneous programming works fine for extremely short pieces (up to 20 or 30 lines of code) or for general experimenting and the testing of a new idea, but it has an extremely negative effect on serious, "finished" programming work. Following such base instincts will lead you to despair and degradation. Your ill-planned code will look like a plate of spaghetti, each line falling over the next, variables scattered willy-nilly, subroutines crashing into each other. You'll end up dressed in tatters, hanging around sidewalks in front of computer stores pestering passersby for scraps of program plans, searching through rubbish barrels for torn pieces of coding outlines. Your only hope is to practice the principles you'll learn here. Don't say I didn't warn you.

### **How to Do This Session**

This session is by far the longest in the tutorial, and it is also perhaps the most important. Take your time with it. The program you'll be developing ties together everything you've learned—and then some. Take several days to do this session if you have to. If you get tired working on this stuff, go do something else. Play around for a while; experiment on your own. Then come back and continue where you left off.

The session is meant to be a review of the whole tutorial. It will help you find where your strong points and your weak points lie. When you come upon a section that asks you to write a piece of code you can't figure out, look at the suggested solution that I give. If you still can't follow what's happening, go back to the section of the tutorial that applies and reread the material. Experiment more with the statements you're having trouble with. Above all, don't get discouraged. Learning to program is fun, but it's also complicated. Now—on to *The Great American Sheep Race*!

## The First Step: Imagine the Outcome

---

One of the best ways to begin a program plan is to determine the program's final outcome. You need to determine what the program is supposed to produce and what the product will look like. You don't have to be rigid about this; after all, you may discover in the course of writing the program that you forgot some vital item (I once wrote a budget program and forgot to allocate money for quarterly estimated taxes—a natural mistake). But you can always add features to a program later; it's the overall design you need to be concerned about.

Usually, if I'm designing a graphics program like the one in this session, I hand-draw my proposed outcomes using the MacPaint application; if the program I'm designing is supposed to produce reports (like my budgeting program, for example), I'll simulate one using MacWrite. Sometimes I'll do several drawings or mock up several reports if the proposed program calls for them.

Figure 12-1 shows you the outcome for *The Great American Sheep Race*; our job is to design and write the program that ultimately gets to that point.

### What the Outcome Implies

Figure 12-1 tells you a lot about the program you're going to write. By looking at the figure you can see that you're going to need certain elements: a race track, a whole flock of racers with identifying numbers, a finish line, an announcement at the end of the race saying which racer wins, and an option for another race.



**Figure 12-1** Ending of *The Great American Sheep Race*.

Certain questions start to come up: How do I move a sheep across the screen? How do I move *all* the sheep across the screen? How do I move them as if they were in a race? How do I get those little numbers to appear on the sides of the sheep? How many racers should there be? Besides the winner's announcement, should there be "track announcements" that say who's ahead during the course of the race? Should there be betting and a tote board? If there is betting, how can I fix the race?

## Locking Down Some Goals: Program Specification

---

Solid program planning has as much to do with exclusion as with inclusion. You have to decide, at least to some extent, on the limits of your program. Professional programmers often work with a document called the **program specification**, a specially prepared report (sometimes a bunch of notes scribbled on a napkin) that describes in varying degrees of detail what a program is supposed to do. The program specification often comes with drawings and/or simulated reports like the ones I talked about earlier.

In larger companies, experts (sort of) with titles like “program analyst” or “program design specialist” write the specifications and give them to teams of programmers to implement. In this case I’m the program design specialist (you can call me “sir”), and you and I are both the programmers. Here’s the program specification for *The Great American Sheep Race*. It’s pretty detailed, but this is your first time working with a formal specification. You can leave more room for spontaneity in your own specifications. Later on, you can change this specification and rewrite the program if you like.

## Overview

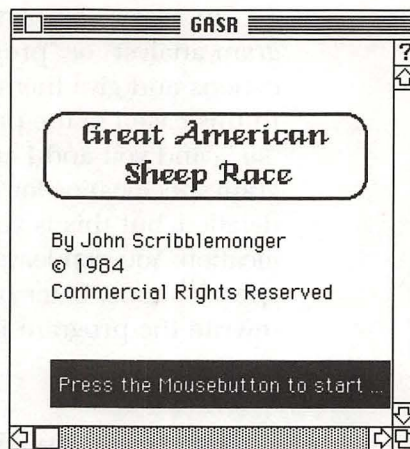
**The Great American Sheep Race** demonstrates character animation on the Macintosh computer, simulating a race among a specific number of contestants. Numbered racers move from starting positions on the left edge of the output window toward the finish line near the right edge of the window, with the number of the leading racer constantly updated and displayed at the bottom. When one racer crosses the finish line, the race ends and that racer is declared the winner. The computer operator then has the option of watching another race.

## Detailed Description

**Opening Message** When the program runs, the computer operator first sees a “title page” (Figure 12-2). When the operator presses the mouse button, this page vanishes and the “field” (Figure 12-3) replaces it.

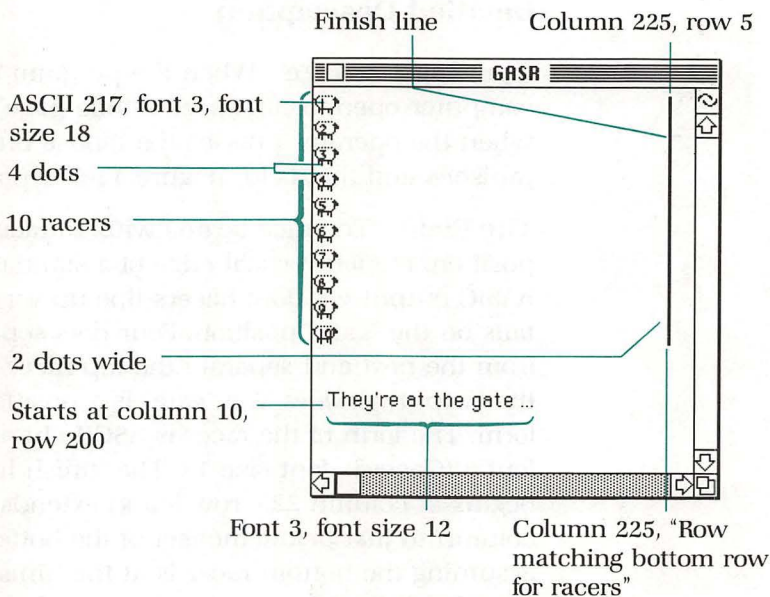
**The Field** The race begins with 10 racers in the “gate” position, the left vertical edge of a standard Macintosh BASIC output window. Racers line up vertically with their tails on the “gate” position. Four dots separate each racer from the next and separate the top racer from the top of the output window. The “gate” is a position only; it has no form. The form of the racer is ASCII character 217 from font 3 (Geneva), font size 18. The “finish line” position begins at column 225, row 5 and extends down the column to just below the feet of the bottom racer, assuming the bottom racer is at the “finish line” position. The “finish line” is a vertical bar two dots wide.





**Figure 12-2** Title page

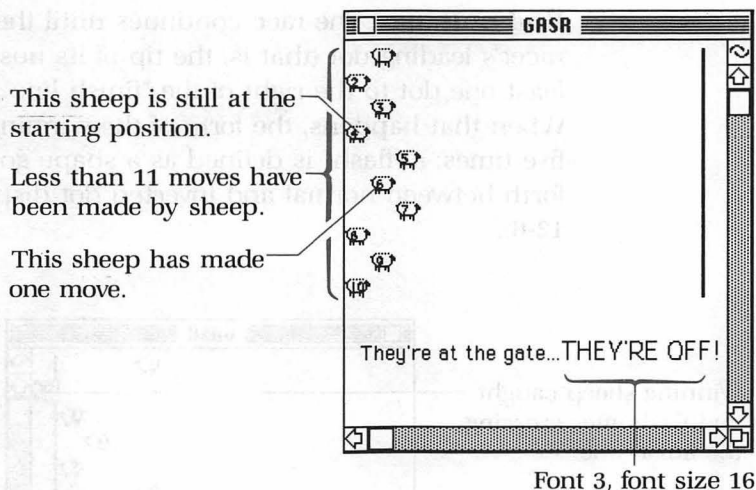
The display remains unmoving in the output window for three seconds or until the operator presses the mouse button. During this time, the message “They’re at the gate...” appears at the bottom right edge of the screen (starting at column 10, row 200) in 12-point Geneva (font 3, font size 12).



**Figure 12-3** The field

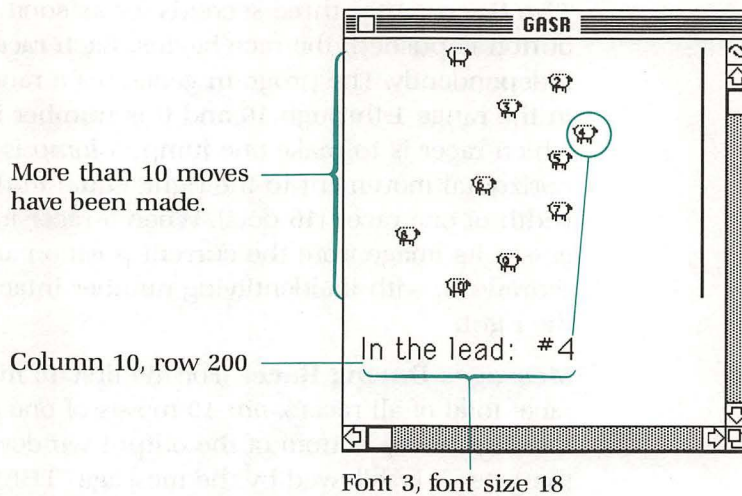
**The Race** After three seconds (or as soon as the mouse button is pushed) the race begins. Each racer moves independently. The program generates a random number in the range 1 through 10 and this number indicates which racer is to make one jump. A *jump* is defined as a horizontal movement to the right, equal in dots to the width of one racer (16 dots). When a racer jumps, BASIC erases its image from the current position and then redraws it, with its identifying number intact, 16 dots to the right.

**Messages During Race** For the first 10 moves of the race (total of all racers, *not* 10 moves of one racer), the message at the bottom of the output window ("They're at the gate...") is followed by the message "THEY'RE OFF!!!" in font 3, font size 16 (Figure 12-4).



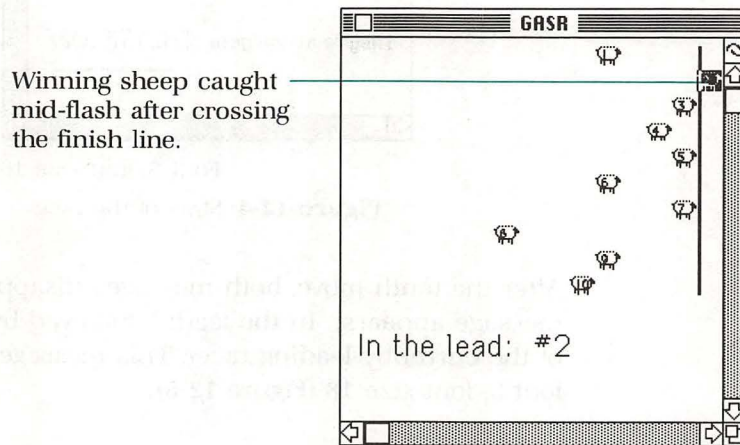
**Figure 12-4** Start of the race

After the tenth move, both messages disappear and a new message appears: "In the lead: ", followed by the number of the currently leading racer. This message appears in font 3, font size 18 (Figure 12-5).



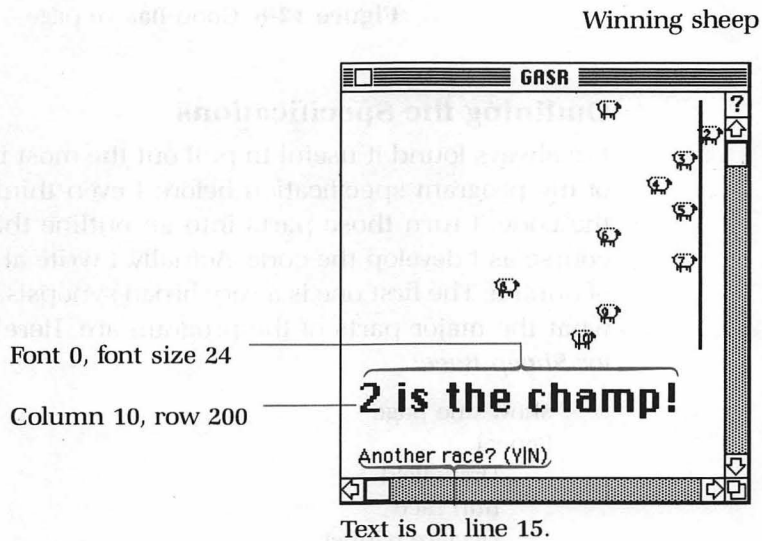
**Figure 12-5** Leader of the pack

**End of Race** The race continues until the leading racer's leading dot (that is, the tip of its nose) is drawn at least one dot to the right of the "finish line" vertical bar. When that happens, the form of the winning racer flashes five times; a "flash" is defined as a shape going back and forth between normal and inverted dot display (Figure 12-6).



**Figure 12-6** Crossing the finish line

Immediately after the flashing, the “In the lead” message at the bottom of the window vanishes. A new message appears at column 10, row 200, reading *w* “ is the champ!”, where *w* is the number of the winning racer; the variable name *w* is arbitrary. The message appears in a font and font size to be determined by the programmer. Immediately after this, another message appears at the window’s bottom reading “Another race? (Y|N)” in standard text (Figure 12-7). If the operator types “y” or “Y”, the program repeats, starting from the section marked “The Field.” If anything else is typed, the output window clears and the “ending page” appears.



**Figure 12-7** The champion sheep

**Ending Page** A racer drawn in font size 164 appears in the middle of an otherwise blank output window. The word “Baa-ye.” shows in place of the racer’s number (Figure 12-8) and the program ends.



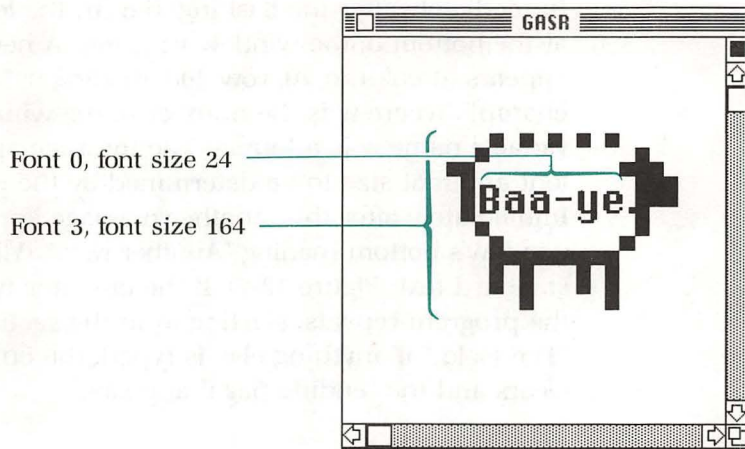


Figure 12-8 Good-Baa-ye page

### Outlining the Specifications

I've always found it useful to pull out the most important parts of my program specification before I even think about writing the code. I turn those parts into an outline that keeps me on course as I develop the code. Actually, I write at least two levels of outline. The first one is a very broad synopsis, telling me only what the major parts of the program are. Here's that synopsis for *Sheep Race*:

```
Show title page
Repeat
    Draw field
    Run race
    Declare winner
    Check go-again
Until done
Show end page
```

My second outline fills in most of the details, with each of the elements of the broad synopsis acting as a major category. As I write this outline, I include references to any tables, figures, or notes from my program specification that might help me write the program. The more detailed outline for the program might look like this:

Show title page (Figure 12-2)

show program title

show author credit

show copyright info

show starting instructions

wait until user presses mouse button

Repeat

Draw field (Figure 12-3)

show 10 numbered sheep

show finish line

show opening message

wait 3 seconds or until user presses mouse button

Run race

repeat

get random number 1 thru 10

move sheep matching random number

display message(s)

if < 11 moves add: "THEY'RE OFF" (Figure 12-4)

else replace: "In the lead: "; Leader (Figure 12-5)

check for winner

until winner

Declare winner

flash winning sheep 5 times (Figure 12-6)

display winning message

Check go-again

display repeat/stop question (Figure 12-7)

wait for repeat/stop instruction

Until user signals end

Show end page (Figure 12-8)

show big sheep

show message on sheep's side

I use this outline as my guide when I write the real code. The outline keeps me on target so that I don't wander too far from my goal. Before I write the code for a section of the outline, I often rewrite that section in even more detail.

## Outline as Pseudocode

---

It's no accident that the outline looks something like a program listing, what with the indentation and the use of terms like "Repeat" and "Until" (which should remind you of variations on a DO\LOOP with EXIT options). This kind of outline, especially when it's written in extreme detail, is called **pseudocode** because its form is relatively close to what eventually will become the program. Here's a more detailed sample of pseudocode, developed from the "Run Race" segment of the outline. It's what you have when you rewrite the outline with one more level of detail (the new material is italicized):

```
repeat
  get random number 1 thru 10
  move sheep matching random number
  find matching sheep's image
  erase old image
  move pen forward one sheep-width (16 dots)
  redraw sheep
  redraw sheep's number on sheep's side
display message(s)
  keep count of moves
  determine who's ahead
  if moves total < 11 print "THEY'RE OFF" (Figure 12-4)
  else
    erase old message
    reset pen position to 10, 200
    display: "In the lead: "; Leader (Figure 12-5)
check for winner
  if leader's nose beyond column 226 then
    leader wins;
    leave the loop
until winner
```

### Pseudocode as Algorithm

Pseudocode shows you what specific steps you have to take to write the actual program lines. In effect, each segment of pseudocode is an **algorithm**—a step-by-step procedure for accomplishing a particular programming goal. For example, the five-line pseudocode segment under “move sheep matching random number” gives each of the steps the program must take to move the proper sheep ahead one move. All you have to do to write the program is translate the pseudocode into real code.

## Writing Code in Modules

---

Detailed pseudocode lets you write code in units called **modules**, with each module doing a particular job. When code is written in small, independent modules, it is easier to design, write, and debug. The approach of writing a general outline and then breaking it down into specific modules is called **top-down programming**.

I suggest that you write the *Sheep Race* program in modules, following the details from the program specification and the pseudocode outline. Write the modules in the suggested order. I'll prepare you for writing each module by giving you tips about what structures to use or how to get BASIC to do stuff you may not know about yet. Then go off to write and debug the code for that particular module. Finally, come back to the tutorial, where you'll see how I wrote the module.





## For Your Information

**For Wizards Only** You can go off on your own entirely if you've found most things in this tutorial easy. The specification, figures, and outline give enough direction so that theoretically you can write all the modules by yourself; but I don't recommend that you do that. Be patient and stick with me for the rest of the session. I give little hints in each section that don't appear anywhere else, and some of the sections have more detailed pseudocode. But you're the boss; just be sure to read my final comments at the end.

### Start with the Title Page

The pseudocode for the title page tells you what steps to take:

```
Show title page (Figure 12-2)
  show program title
  show author credit
  show copyright info
  show starting instructions
  wait until user presses mouse button
```

The figure that goes with it (Figure 12-2) shows you how the page should look. The only details missing are the row and column coordinates for the displayed messages, the instructions for drawing the rounded rectangle around the title, and the ones for putting the inverted rectangle over the starting instructions. You can handle all those OK. You get the copyright symbol by pressing Option-g.

OK, now go write the code. Make it a subroutine named `Title.Page`.

**Here's My Title.Page** To test this subroutine, I set up a little program that reads:

```
GOSUB Title.Page:
END PROGRAM
```

You can use this pattern to test any subroutine without getting the error message "RETURN Without GOSUB." Alternately, you could leave the RETURN statement off the subroutine until later—but if you forget to put it back in, you're in trouble:

Title.Page:

```
SET PENSIZE 2, 2    ! Make the penpoint thick
FRAME ROUNDRECT 20, 45; 225, 100 WITH 30, 30
SET PENSIZE 1, 1    ! Put it back to "normal"
```

```
SET FONTSIZE 18
SET FONT 5
SET PENPOS 45, 65
GPRINT "Great American"
SET PENPOS 72, 90
GPRINT "Sheep Race"
```

```
SET FONTSIZE 12
SET FONT 3
SET PENPOS 25, 130
GPRINT "By John Scribblemonger"
GPRINT "© 1984"
GPRINT "Commercial Rights Reserved"
```

```
SET PENPOS 30, 220
GPRINT "Press the Mousebutton to start..."
INVERT RECT 25, 200; 240, 230
```

```
BTNWAIT
RETURN
```

After I wrote this routine, I saved it on a disk under the name *Title.Page*. I suggest that you, too, save each module separately on the disk. Later on, after you've written all the routines, you can call them back one by one and paste them all together. I'll show you how at the proper time.

### **Next: Drawing the Field**

This segment is a little trickier; it makes sense to cover it in subsections.

```
Draw field (Figure 12-3)
  show 10 numbered sheep
  show finish line
  show opening message
  wait 3 seconds or until user presses mouse button
```

To draw 10 numbered sheep, first figure out how to draw one sheep. Then you can use a loop of some kind to draw all 10 at once. Forget about sheep numbers for now.

You need to know that each sheep is 12 dots high and that the program specification calls for a 4-dot separation between sheep. Put this total (that is, 16) in a variable called `Sheep.Height`.

Here's some hints to make your job easier: use variables for the column and row positions of the pen; call the variables `Row` and `Column`, and set `Row` to hold the value of `Sheep.Height`. Go do it.

**My Single Sheep** I begin my routine by creating a string variable called `Sheep$`. I'm going to use this variable throughout the program to hold the value `CHR$(217)`, the ASCII code that in font 3, fontsize 18 produces the sheep character. `Sheep$` reminds me of a sheep far more than does `CHR$(217)`. Here's the code:

```
Sheep$ = CHR$(217)
Sheep.Height = 16
Column = 0
Row = Sheep.Height

SET FONTSIZE 18
SET FONT 3
SET PENPOS Column, Row
GPRINT Sheep$
```

I set the variable `Column` to 0, just as Figure 12-3 showed.

It's not hard to modify this code so that it displays 10 sheep going down the display at the left edge, as the specifications demands. Try it—even if your own code already does it.

**My Ten Sheep** All I had to do to this code was change one line and add two more lines. The new code is italicized:

```
FOR Racer = 1 TO 10
  Row = Sheep.Height * Racer ! Tricky, no? Think about it.
  SET FONTSIZE 18
  SET FONT 3
  SET PENPOS Column, Row
  GPRINT Sheep$
NEXT Racer
```

Now for the numbers. You need to use a font size of 7 for these little numbers to make them fit properly. To place a number correctly on a sheep's side, set the pen three dots to the right and three dots up from where you start drawing the sheep. (Remember that the character you draw has its left bottom dot at the stated PENPOS.) Again, even if you've already figured out how to draw all 10 sheep with their numbers in place, modify my code to do it. You just need to add three lines to my code.

**My Numbered Sheep** Once more, the new code is italicized. As you can see, I took advantage of the loop structure already written to do most of the work:

```
FOR Racer = 1 TO 10
  Row = Sheep.Height * Racer
  SET FONTSIZE 18
  SET FONT 3
  SET PENPOS Column, Row
  GPRINT Sheep$

  SET FONTSIZE 7
  SET PENPOS Column + 3, Row - 3
  GPRINT Racer
NEXT Racer
```

Now that the sheep are taken care of, it's time to draw the finish line and the opening message. The first task is only slightly complex; all the information you need is in Figure 12-3. Here's a hint: after you've run the code listed above, you'll already have a variable holding the value you need to set the bottom position of the finish line. The opening message is a piece of cake. Go write code; add it to the bottom of the code you've already written in this module.

**My Finish Line and Message** The finish line parameters call for a vertical bar two dots wide extending from column 225, row 5 to column 225, row whatever-is-the-bottom-of-the-lowest-sheep. Here's my code:

```
SET PENSIZE 2, 1
PLOT 225, 5; 225, Row
```

When you run this code segment, including the sheep-drawing part, the variable Row already contains the bottom row location for the bottom sheep, because—as I said earlier—a character is drawn from the ground up.



The position and wording for the opening message is so clearly spelled out in Figure 12-3 that the code practically writes itself:

```
SET PENPOS 10, 200
SET FONTSIZE 12
GPRINT "They're at the gate...";
ASK PENPOS Message.Column, Message.Row
```

You don't have to set font 3; it's already set. The only problem you're likely to have with this code is understanding why I've added that last line. It's there because the pseudocode calls for adding a message later that will come hard on the heels of the "gate" message. I know that the message will appear in row 200, but I don't know where the last dot in "They're at the gate..." falls. This line stores the current pen position—which is just after the last dot in "gate..."—in the variables following the keyword PENPOS. Later you'll see me using the values I capture here in the Display.Messages portion of the "Run Race" module.

Before tackling the rest of this segment, save this part under the name *Field*. Before you store it on the disk, start it with the proper subroutine label "Field:", and end it with a RETURN.

Here's what the whole module looks like:

Field:

```
Sheep$ = CHR$(217)
Sheep.Height = 16
Column = 0

FOR Racer = 1 TO 10
    Row = Sheep.Height * Racer
    SET FONTSIZE 18
    SET FONT 3
    SET PENPOS Column, Row
    GPRINT Sheep$

    SET FONTSIZE 7
    SET PENPOS Column + 3, Row - 3,
    GPRINT Racer
NEXT Racer
```

```

SET PENSIZE 2, 1
PLOT 225, 5; 225, Row
SET PENPOS 10, 200
SET FONTSIZE 12
GPRINT "They're at the gate...";
ASK PENPOS Message.Column, Message.Row
RETURN

```

### Working Out the Timer Module

The section of the outline that calls for a three-second delay is sufficiently complex and different from the rest of the code to call for its own separate module. As soon as you've written it, save it on a disk.

First, you need to expand the outline if you're going to use it as pseudocode. Allow me:

```

set a variable to current time
repeat
  if a second has passed then
    increment counter;
    reset variable to current time
  if the counter holds 3 then leave loop
  if mouse button is pressed then leave loop
until done

```

Here's a reminder: TIME\$ changes its value every second. Don't be thrown by this module; look at the pseudocode and see what it suggests. Then write the code.

**My Timer** This module is an especially useful one. Lots of programs need timers; you can use this same module in all kinds of situations in addition to the one you're working on:

Timer:

```

Oldtime$ = TIME$           ! Get original time
DO
  IF TIME$ <> Oldtime$ THEN  ! Has a second passed?
    Seconds = Seconds + 1    ! Increment counter
    Oldtime$ = TIME$        ! Reset the time
  ENDIF
  IF Seconds = 3 THEN EXIT   ! Been in loop 3 seconds yet?
  IF MOUSEB~ THEN EXIT      ! Mouse button pressed?
LOOP
Seconds = 0                 ! Reset counter to 0
RETURN

```

I reset the counter “Seconds” to 0 before leaving the subroutine. If I happen to use that counter again, I don’t want it giving me any false readings based on leftover values.

There are lots of ways to write this code; if your way is different and it works, that’s great! Be sure to save the code when you’re done; I’ve used the name *Timer* for this section.

### **Run Race: The Business End of the Program**

This section of the pseudocode describes the heart of the program. Here’s what the whole pseudocode block looks like:

```

Run race
  repeat
    get random number 1 thru 10
    move sheep matching random number
    find matching sheep's image
    erase old image
    move pen forward one sheep-width (16 dots)
    redraw sheep
    redraw sheep's number on sheep's side
  display message(s)
  keep count of moves
  determine who's ahead
  if moves total < 11 print "THEY'RE OFF!" (Figure 12-4)
  else
    erase old message
    reset pen position to 10, 200
    display: "In the lead: "; Leader (Figure 12-5)
  check for winner
  if leader's nose beyond column 226 then
    leader wins;
    leave the loop
  until winner

```

This block is made up of several sections. The major section headings are:

```

get random number 1 thru 10
move sheep matching random number
display messages
check for winner

```

Your job will be a lot easier if you do each of these four subsections separately.

First, write the random number part. This section determines which sheep moves forward one sheep-width. You won't need my help here; it's simple to do. After you've written and tested your code, come back to see the way I did it.

**My Random Number Maker** Now, here's a simple piece of code! If yours is different, be sure it works before going on:

```
Move = INT(RND(10)) + 1
```

One way to make sure the code is working the way you expect it to work is to have it display 50 or 60 numbers and test to make sure that none is less than 1 or more than 10. I used this method:

```
FOR Test = 1 TO 50  
    Move = INT(RND(10)) + 1  
    PRINT Move  
NEXT Test
```



### For Your Information

By the way, the names for your subroutines and your variables need not be the same as mine; but later on when I pull all of these different routines together into a program, I'll be using the names I assign to them as I go along. For that reason it might be easier for you if your names are the same as the ones I use. Now on to the Sheep Moves section!

**Moving the Sheep** Moving a sheep involves several steps. You know which sheep to move; its number is the same as the random number generated by the line you just wrote. Now you have to determine where the sheep is on the field, erase its old image, and redraw its new image one move forward. Here's the pseudocode, expanded even further than before. Again, I've italicized the newest parts:



```

Recall sheep Move's stored old position
erase old image
    erase a rectangle containing the sheep's image
redraw sheep
    move pen forward one sheep-width (16 dots)
    Row = sheep-to-move times Sheep.Height
    New.Column = Old.Column + 16 dots
    store this position in case sheep moves again later
redraw sheep's number on sheep's side

```

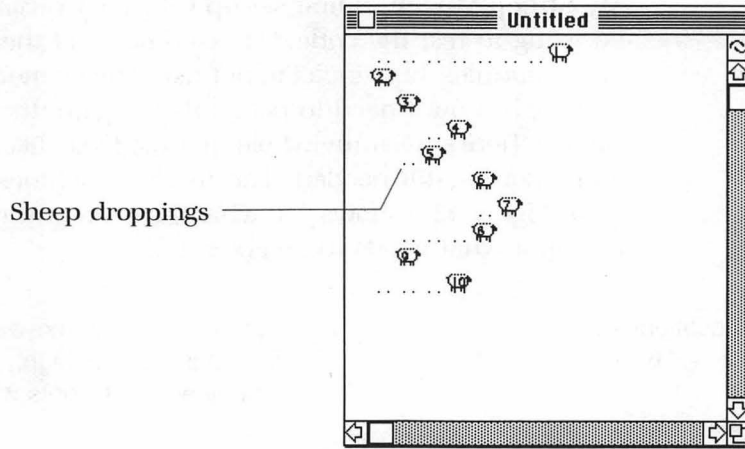
There's a whole new concept here: the idea of a stored old position. Before you can move a sheep, you have to know where it is. On the first move, it's easy: all sheep start off at the left edge of the output window, column 0. After that, things get a little more complicated. Here's a hint: you need to know a sheep's width and height in dots so that you can properly place the pen to either erase or redraw it. Remember that a sheep "lives" in a square 16 dots wide by 16 dots high, including a four-dot buffer between the top of its head and the foot of the sheep above. Thus, once a sheep has moved, it's 16 dots to the right of its old position. You need some variable in which to store a sheep's movements.

Since you have 10 sheep, you'll need 10 variables, one for each sheep. Looks like a perfect spot for an array; that way the same variable name can be used for all 10 sheep. For example, `Sheep.Location(5)` would always hold the column location of Sheep #5's leftmost dot. You don't have to store the row position because it's always the product of a sheep's height times this sheep's position number. Assuming you're using the variable name `Move` for the next sheep to move, the expression for the proper row would be `Move * Sheep.Height`.

Be careful here, though. Before you use any array, you must first use `DIM` to dimension it; and once you've dimensioned it, you can't dimension it again without getting an error message. Thus you should put your `DIM` statement at the very start of the program. For now, though, since you're building separate modules that you won't bring together until later, just put your `DIM` statement at the start of this module.

Try your hand at writing a `Sheep.Mover` module. Don't forget to test it!





**Figure 12-9** Sheep droppings

The line of code that erases a sheep's old image needs further explanation:

```
ERASE RECT Column, Row - Sheep.Height; Column + Sheep.Width, Row
```

When you're forming a shape, you need to specify the upper left coordinate and then the lower right coordinate. But MacBASIC draws its characters (including the sheep) from the ground up, the opposite of the way it draws its shapes. Assume that the sheep you want to erase is Sheep #4. Sheep #4 is "built" starting with row 64—16 times 4. But its "top" is 16 dots above that, so you have to subtract 15 dots from Row to find the correct starting row for ERASE RECT. Thus the correct expression for the first coordinate is Column, Row - Sheep.Height.

(Figure 12-10). If that isn't clear, the best thing to do is to type in my code and experiment by changing the ERASE RECT line in various ways. Your own experience will teach you better than my deathless prose ever can.

**About the Sheep Droppings** If you were to run this code as it stands (which you are invited, although not impelled, to do) you'd see the sheep galloping off into oblivion. You'd also see left behind what appears to be sheep droppings, clearly visible in Figure 12-9. Actually, those extra dots are the bottom dots of the sheep's feet. The problem is in the ERASE RECT line and has to do with a technical detail about the way shapes work

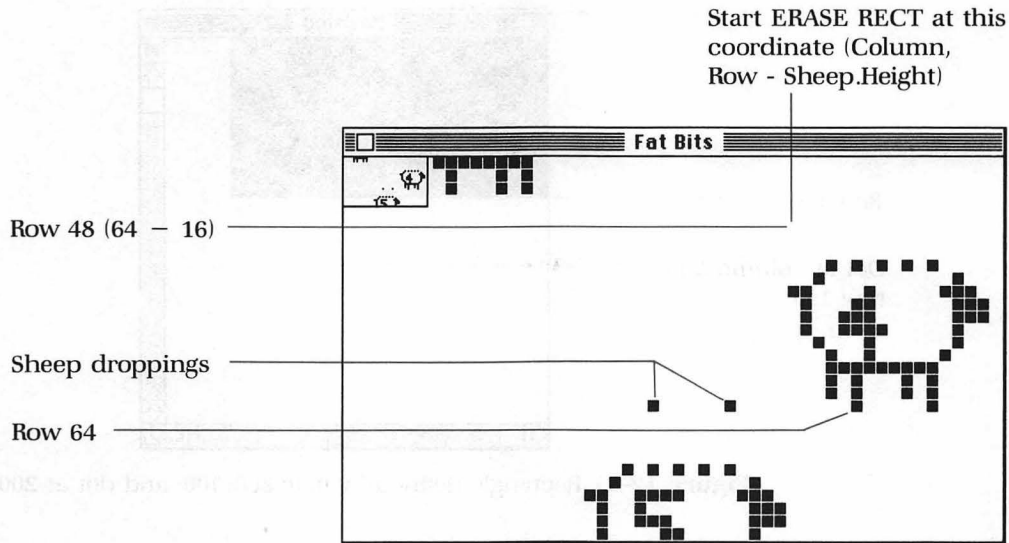


Figure 12-10 Close-up of sheep

(rather than with whatever disgusting personal habits sheep may or may not display). When you tell BASIC to plot a point with the statement `PLOT 200, 100`, it actually turns on the dot with the coordinates column 200, row 100. However, when you tell BASIC to paint a rectangle from coordinate 0,0 to coordinate 200, 100, it paints the rectangle from 0, 0 to column 199, row 99. Try this program to see it happen; Figure 12-11 shows you the result:

```
PAINT RECT 0, 0; 200, 100
BTNWAIT
PLOT 200, 100
```

When you run the program, look at the lower-right corner of the rectangle. Then press the mouse button and see the new point plotted.

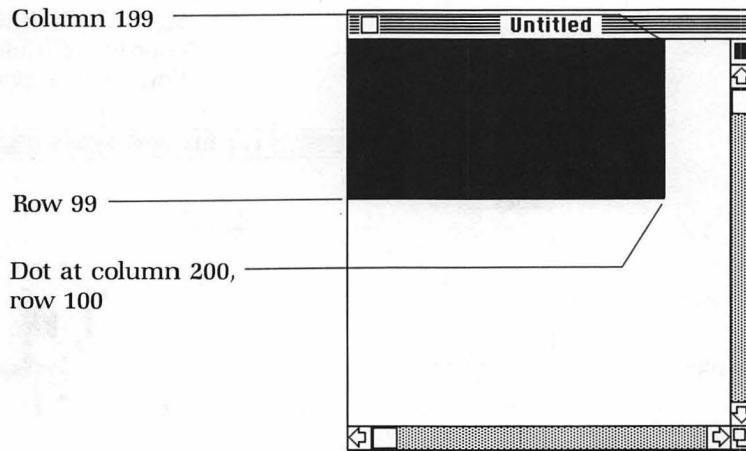
What's true for painted rectangles is also true for erased rectangles. In the statement

```
ERASE RECT Column, Row - Sheep.Height; Column + Sheep.Width, Row
```

the *Row* part of the second coordinate is actually one pixel less than it needs to be to completely erase the old sheep.

One way to clean up the act of these uncivilized wool makers is to increase *Row* for the lower-right corner by 1:





**Figure 12-11** Rectangle defined by 0, 0; 200, 100; and dot at 200, 100

```
ERASE RECT Column, Row - Sheep.Height; Column + Sheep.Width, Row + 1
```

Make sure you have your working, debugged Sheep.Mover module stored on disk. Then go on to read about creating the Display Messages block.

### Adding the Racing.Messages Block

Most of this block is really easy stuff. The only thing that might give you trouble is figuring out which sheep is in the lead. Here's the pseudocode:

```
keep count of moves
determine who's ahead
if moves total < 11 print "THEY'RE OFF!" (Figure 12-4)
else
  erase old message
  reset pen position to 10, 200
  display: "In the lead: "; Leader (Figure 12-5)
```

Keeping count of the moves is easy; each time a sheep jumps ahead one move, just increment a counter. If less than 11 moves have been made, print one message; if 11 or more moves, clear the message area and display the other message.

But how do you keep track of who's ahead? See if you can work it out on your own. Here's a hint: I'm using a new variable, Leader, to hold the number of the sheep that's currently ahead.

Each time a sheep makes a move, I compare its new position to the position of the leading sheep.

Go write the code. Take your time, and be sure to test the code after you've written it to make sure it works.

**My Racing.Messages** I'm naming this subroutine Racing.Messages to distinguish it from other messages that might appear. Here's the code:

```
Racing.Messages:
  Jump = Jump + 1      ! Keep track of all the moves
  IF Sheep.Location(Move) > Sheep.Location(Leader) THEN Leader = Move

  IF Jump < 11 THEN    ! Do this stuff if within first 10 moves
    SET PENPOS Message.Column, Message.Row
    SET FONTSIZE 16
    GPRINT "THEY'RE OFF!"
  ELSE                ! Do this stuff if beyond 10th move
    ERASE RECT 10, 180; 250, 200
    SET PENPOS 10, 200
    SET FONTSIZE 18
    GPRINT "In the lead: #"; Leader
  ENDIF
RETURN
```

The only tricky piece here is the second line:

```
IF Sheep.Location(Move) > Sheep.Location(Leader) THEN Leader = Move
```

Translated into the Mother Tongue, that says "If the column position of the sheep that just moved is greater than the column position of the sheep formerly in the lead, establish the sheep that just moved as the new lead sheep." If the way I've written that code line is confusing to you, put the following lines in its place:

```
This.Sheep = Move
This.Sheep.Position = Sheep.Location(Move)
Leader.Position = Sheep.Location(Leader)
IF This.Sheep.Position > Leader.Position THEN Leader = This.Sheep
```

The result is the same, but the code reads a little more clearly.

You might also be wondering where I got the coordinates to use in the ERASE RECT line. Essentially, they're just rough estimates. I set the upper left coordinate to start below the bottom of the lowest sheep (anywhere beyond 10 times 16 for the row is OK—10 sheep times 16 dots for each) and to fall in

a column to the left of the first letter (set at column 10); the lower right coordinate is 40 dots down and well past the right edge. I didn't know where the "THEY'RE OFF!" message would end and didn't want to take the time and energy to find out (programming is like that sometimes). This estimating, erase-by-brute-force method is crude but effective.

### **The Test.For.Winner Module**

The last section of the code for the "Run Race" group of modules is very easy (famous last words). Reading the pseudocode, you're reminded that the whole race is in a loop; the sheep keep on moving until there's a winner:

```
if leader's nose beyond column 226 then
    leader wins;
    leave the loop
```

You already know where the leader's tail is; you'll find its column position in `Sheep.Location(Leader)`. You also know that a sheep's width is 16 dots (stored in the variable `Sheep.Width`). That means it's a matter of simple arithmetic to find out where the sheep's nose is. If the nose is beyond column 226, then you've got a winner. If you do have a winner, then set a flag—that is, set some variable as a signal that BASIC can later interpret (we'll write the "interpreting" code later). I'm using the boolean variable `Winner` as my flag variable.

Now write the code. Don't worry about where to put the code in the program; just write it as a subroutine for now.

**My Test.For.Winner Routine** To find out if there's a winner yet, I compare the position of the leader's nose to the position of the finish line. If the nose is beyond the finish line, I set a flag that the program can check later:

```
Test.For.Winner:
    Nose = Sheep.Location(Leader) + Sheep.Width
    IF Nose > 226 THEN Winner = true
    RETURN
```

### **The Declare.Winner Block**

You're rapidly moving towards the end of the code now; there are only a few major blocks left. The first block deals with declaring the winner. Here's the pseudocode:

```
flash winning sheep 5 times (Figure 12-6)
display winning message
```

Take the “flash” part first. The way to get something to “flash” is to switch back and forth between a normal display (black characters against a white background) and an inverted display (white characters against a black background). One way to do that is to use INVERT RECT in a loop of some kind. The rectangle to invert in this case should include only the block of dots taken up by the sheep character. Hint #1: We’ve been referring to the height of a sheep as 16 dots, but the top four dots are just space. Hint #2: As you saw earlier in this session, the bottom coordinate of a rectangle is one dot less than the bottom coordinate of a sheep character. Hint #3: If the flashing goes by too quickly, put an empty “delay loop” within the major “flash” loop to slow things down; I use the form

```
FOR Stall = 1 TO 500
NEXT Stall
```

Now for the winning message. Put it down at the bottom of the display, replacing the “In the lead” message. You can choose your own font and font size. You’ll probably want to erase the message that’s already down there before you write the new message. Hint: Try reusing the code you already wrote in the Racing.Messages module.

Write the code now. Be sure to test your code before looking at the way I wrote the block. You may have to combine some of the modules you’ve already written to give this part a good test. Yet Another Hint: To combine modules stored on a disk, use the Cut and Paste commands (more on this later).

You don’t have to worry yet about how the program gets to this routine; as I said earlier, we’ll pull all this stuff together later.

**My Declare.Winner Module** To do the “flash” section, I first set up some additional variables for convenience. I found that the lines were getting too long and complex to read clearly, and I didn’t want to bother enlarging the window or scrolling the text to see all the code. It’s a lot easier to read (and thus to debug) this way:

Declare.Winner:

```

Tail = Sheep.Location(Leader)      ! starting column coordinate
Head = Leader * Sheep.Height - 12  ! starting row coordinate
Foot = Leader * Sheep.Height + 1   ! ending row coordinate
                                   ! ("Nose" is ending column)

FOR Flash = 1 TO 10
  INVERT RECT Tail, Head; Nose, Foot
  FOR Stall = 1 TO 500              ! Here's the delay loop
    NEXT Stall
NEXT Flash

ERASE RECT 10, 180; 250, 220        ! Lifted from "Racing.Messages"

SET PENPOS 10, 200
SET FONTSIZE 24                     ! Big type size
SET FONT 0                          ! Bold, thick letters
GPRINT Leader; " is the champ!"
RETURN

```

### The Go.Again Module

This module's code comes into play after a sheep wins the race. Here the program asks if the person operating the computer wants to see another race. If the answer is yes, the program runs another race; if the answer is no, the program goes on to the End.Page module. Here's what the pseudocode calls for:

```

display repeat/stop question (Figure 12-7)
wait for repeat/stop instruction

```

The program specification and Figure 12-7 are both clear on what BASIC should display; I'll leave it to you to write that code. There's only one tough part about the "repeat/stop instruction": How does BASIC leave this module and let the program know what it's supposed to do next? I suggest you use a flag again. If the operator wants to see another race, set a variable (I'm using Repeat) to hold the value *true*; if he or she has seen enough sheep, set the variable to hold *false*.

Here's a final hint: The specs call for "standard text" at the bottom of the window to ask the "Another race?" question; I'd use SET VPOS 15 to get the insertion point down to the bottom.

**What VPOS Does** BASIC keeps graphic and nongraphic information separate to some extent. It sees any text that's part of a PRINT or an INPUT statement as nongraphic material. The



insertion points for graphic and nongraphic material are also separate. Thus the graphic pen position used by GPRINT and the various shapes (PAINT RECT and so on) can be at, say, column 100, row 20, while the nongraphic insertion point used by PRINT and INPUT can be at the start of the 15th text line (left edge of the window, near the bottom). If you want the nongraphic insertion point to be someplace, you'd better put it there explicitly with **VPOS**. This statement moves the insertion point down to whatever PRINT text line you want; SET VPOS automatically multiplies the height of the text (in dots) by the line you set. Line 15 is near the bottom of the output window (15 times 16 dots)—the equivalent of PENPOS 8, 240 for a GPRINT line.

Write that module!

**My Go.Again Module** The only unusual thing in my code is a provision for people who might type a lowercase “y” instead of uppercase “Y” to mean “yes.” I use a SELECT construct to do it:

```
Go.Again:
  SET VPOS 15
  INPUT "Another race? (Y|N) "; Answer$
  SELECT Answer$
    CASE "Y", "y"
      Repeat = true
    CASE ELSE
      Repeat = false
  END SELECT
  RETURN
```

The code just sets a flag that some other BASIC code will interpret. I'll show you how it works when we pull all the code together—after you write the End.Page module.

## The End.Page Module

The End.Page pseudocode is pretty clear:

```
Show end page (Figure 12-8)
  show big sheep
  show message on sheep's side
```

The only thing the program specification doesn't tell us is where in dot coordinates to put the sheep and the sheep's parting message. I'll give you those to save you the time of figuring it out:

Sheep position: column 65, row 160

Message position: column 84, row 110

I'll give you one more hint: Don't forget to erase the screen as soon as you start the module. Go write the code!

**My End.Page Module** The code couldn't be any more straightforward:

```
End.Page:
CLEARWINDOW
SET PENPOS 65, 160
SET FONTSIZE 164
GPRINT Sheep$;    ! You figure out why we need the
                  !   semicolon.

SET PENPOS 84, 110
SET FONTSIZE 24
SET FONT 0
GPRINT "Baa-ye."; ! Try this without a semicolon.
RETURN
```

### Bringing the Modules Together

What follows is a synopsis of the code you've written so far. You'll need to keep it in mind as you construct *The Great American Sheep Race* out of its component parts. First, here's a list of all the code modules. You may have used different names, but the action should be the same:

- **Title.Page:** shows the name of the program and gives credits.
- **Field:** shows the racers on the track and gives a starting message.
- **Timer:** delays the program 3 seconds or until operator presses the mouse button.
- **Sheep.Mover:** moves a random sheep ahead by one sheep-width.
- **Racing.Messages:** lets operator know who's ahead.
- **Test.For.Winner:** checks to see if any sheep has crossed the finish line.

- **Declare.Winner:** flashes the winning sheep and says “x is the champ!”
- **Go.Again:** checks to see if operator wants to watch another race.
- **End.Page:** shows a giant sheep saying good-bye.

Two of these modules set flags for you: `Test.For.Winner` assigns the value *true* to variable `Winner` when a sheep crosses the finish line; `Go.Again` puts *true* in variable `Repeat` if operator wants to watch another race.

Looking back at both the program specifications and the more detailed pseudocode, you can separate the modules into groups based on what repeats when.

### **The Module Groups: How They All Work**

- The `Title.Page` module is used only when you first execute the program.
- `Field` and `Timer` go together; they get used just before the start of every race.
- During the progress of the race, the three modules `Sheep.Mover`, `Racing.Messages`, and `Test.For.Winner` constantly repeat in the order given until some sheep crosses the finish line, setting the flag `Winner`.
- The two modules `Declare.Winner` and `Go.Again` happen at the end of each race. `Go.Again` gets a value for the flag `Repeat`; if the value is *true*, then all the modules beginning from `Field` are repeated. If the value is *false*, the program goes on to the next module.
- The last module is `End.Page`; it happens if the operator is done watching the mutton march, as signaled by the variable `Repeat` holding *false*.

Using this description, I can construct new, highly compact pseudocode to show how to convert the separate modules into a running program:

Title.Page:

Repeat

Field:

Timer:

Repeat

Sheep.Mover:

Racing.Messages:

Test.for.Winner:

Until Winner Found

Declare.Winner:

Go.Again:

Until User Done

End.Page:

Keeping in mind that all of the modules are actually sub-routines, convert my pseudocode into a block of real code that will control program execution. This new block will come at the start of the program and will direct program flow. Make sure you type your new code into a new, empty output window.

**What My Controlling Block Looks Like** Here's what my controlling block looks like. Yours should look pretty similar.

GOSUB Title.Page:

DO

GOSUB Field:

GOSUB Timer:

DO

GOSUB Sheep.Mover:

GOSUB Racing.Messages:

GOSUB Test.For.Winner:

IF Winner THEN EXIT

LOOP

GOSUB Declare.Winner:

GOSUB Go.Again:

IF NOT Repeat THEN EXIT

LOOP

GOSUB End.Page:

END PROGRAM

## Retrieving the Blocks

The time has come to pull all the blocks together. By using commands from the Edit and File menus you can easily get back all the blocks you've written and stored on the disk. Here's the procedure:

1. Clear away all windows from your desktop except for the one containing the controlling block module that you just wrote.
2. Retrieve the module called Title.Page from the disk.
3. Fix any syntax errors that BASIC might complain about during the retrieval process.
4. Use ⌘ A to select the entire contents of the window.
5. Use ⌘ X to move the contents to the Clipboard.
6. Click the close box to get rid of the module's now-empty window.
7. Use ⌘ V to paste the contents of the Clipboard into the controlling block module's window. Make sure that there's at least one blank line between the last word in the window (it should be the keyword RETURN) and the insertion point. Make sure the insertion point is at the left edge of the window.
8. Retrieve the the module called Field.
9. Repeat steps 3 through 7.
10. Retrieve the rest of the modules in the same way.
11. Store this collection of modules under the name *GASR*.

## The Initializing Routines

Before the program will run properly, you need to move some variables and the DIM Sheep.Location(10) statement to a yet-to-be-written module called Initializers. Additionally, you need to be sure that all the flags and/or counters are cleared of their old values before each race is run; to do that, you'll create another block called ClearOut. To see that what I'm saying is indeed true, run the program as it stands and see what happens. Make sure you answer "y" at the end of the first race to see the race run again.

Run the code in its present form; chances are that it won't get too far. Note everything bad that happens.



**Cleaning Up the Code** Unless your code is radically different from mine, you should have experienced some real messes. First, the title page didn't erase; it hung around all through the race. Then, when you tried to run a second race, you probably got a "redimed array" error message. If you got past that, the second running probably stopped almost instantly. Clearly, you need to do more work on the code.

Don't panic yet; you can fix the whole mess in only a few minutes. You can divide up the problems into two areas: improper initialization and failure to clear variables. I won't ask you to devise this "cleanup" code on your own; I'll show you how to do it.

**Initializing Variables** Nearly all programs need to have certain variables set up at the start of the program. Some of these keep their original values throughout the program; establishing them over and over in different program parts just wastes memory and, in some cases (as with a DIM), stops your program dead. Here's a routine that your program must execute right after Title.Page. Add it now:

Initializers:

CLEARWINDOW	! Get rid of title page
RANDOMIZE	! Scramble random numbers
DIM Sheep.Location(10)	! Set up array for sheep locations
Sheep\$ = CHR\$(217)	! Set up sheep character
Sheep.Height = 16	! This is constant throughout program
Sheep.Width = 16	! Also a constant
RETURN	

The call to this module happens outside of any loops, as does the call to the Title.Page module; you want to execute this code only once each time you run the program.

Use the What to find and Find commands from the Search menu to locate and remove any redundant occurrences of these statements (at least two are in the Sheep.Mover module, and there's a DIM statement roaming around somewhere).

**Clearing Flags and Counters** When you get through watching a race, the program gives you the option of watching another race. But before these rambling rams can again amble across your display, certain things have to happen. You need to clear the screen, reset certain flags, and so on. Here's a block of code to do it; add it to your program.

```

Clear.Out:
  CLEARWINDOW                ! Get rid of old race display
  Jump = 0                   ! Clear dead counter
  Winner~ = false            ! Clear dead flag
  Leader = 0                  ! Clear old winner
  FOR Clearout = 1 TO 10     ! Clear old sheep positions
    Sheep.Location(Clearout) = 0
  NEXT Clearout
RETURN

```

Your code should call this module if the variable Repeat~ holds *true* (see the italicized section of the final controlling code block, below).

Try clearing just one or two of these variables and watch the effect. If you can't figure out why something is happening, go back over the original code that set or changed a particular variable. Experiment by changing the code and watching the results until you follow what's going on.

**The Final Controlling Block** Here's what the controlling code block should look like. I've italicized the two parts that are new. Note the END statement at the bottom to prevent the program flow from accidentally falling into the subroutines. No executable code should precede this block in the program; the only thing above it should be comment statements, all of which begin with the character "!":

```

GOSUB Title.Page:
GOSUB Initializers:
DO
  GOSUB Field:
  GOSUB Timer:
DO
  GOSUB Sheep.Mover:
  GOSUB Racing.Messages:
  GOSUB Test.For.Winner:
  IF Winner~ THEN EXIT
LOOP
  GOSUB Declare.Winner:
  GOSUB Go.Again:
  IF NOT Repeat~ THEN EXIT ELSE GOSUB Clear.Out:
LOOP
GOSUB End.Page:
END PROGRAM

```

## Final Testing and Commenting

Now it's time to do final cleanup and to add whatever needs to be added. First, run the program a number of times to make sure there aren't any bugs in it. Because you did so much planning before you wrote the code, you shouldn't find many bugs, and any that you do find, you should be able to track down and fix easily.

After you've made sure the program runs OK, go back and add comments to the various modules where the code isn't absolutely clear. Remember to add some comments above the controlling code block, giving the name of the program and other vital information.

Be sure to save at least two copies of the program, one each on two separate disks; after all this work, you want to protect yourself from potential disasters.

Here's a complete listing of my version of the program:

```
! Program: Great American Sheep Race
! Code by Scot Kamins
! Copyright 1984 Technology Translated Inc.

! Written as Session 12 to Introduction to Macintosh BASIC
! Version 1.4   June 6, 1984

GOSUB Title.Page:
GOSUB Initializers:
DO
  GOSUB Field:
  GOSUB Timer:

  DO
    GOSUB Sheep.Mover:
    GOSUB Racing.Messages:
    GOSUB Test.For.Winner:
    IF Winner THEN EXIT
  LOOP

  GOSUB Declare.Winner:
  GOSUB Go.Again:
  IF NOT Repeat THEN EXIT ELSE GOSUB Clear.Out:

LOOP
GOSUB End.Page:
END PROGRAM
```

*(Listing continued on next page.)*

Title.Page:

```
SET PENSIZE 2, 2      ! Make it thick
FRAME ROUNDRECT 20, 45; 225, 100 WITH 30, 30
SET PENSIZE 1, 1      ! Reset it
```

```
SET FONTSIZE 18
SET FONT 5
SET PENPOS 45, 65
GPRINT "Great American"
SET PENPOS 72, 90
GPRINT "Sheep Race"
```

```
SET FONTSIZE 12
SET FONT 3
SET PENPOS 25, 130
GPRINT "By John Scribblemonger"
GPRINT "© 1984"
GPRINT "Commercial Rights Reserved"
```

```
SET PENPOS 30, 220
GPRINT "Press the Mousebutton to start..."
INVERT RECT 25, 200; 240, 230
```

```
BTNWAIT
```

```
RETURN
```

Initializers:

```
CLEARWINDOW          ! Get rid of title page
RANDOMIZE              ! Scramble random numbers
DIM Sheep.Location(10) ! Set up array for sheep locations
Sheep$ = CHR$(217)    ! Set up sheep character
Sheep.Height = 16     ! This is constant throughout program
Sheep.Width = 16      ! Also a constant
```

```
RETURN
```

Field:

```
Column = 0

FOR Racer = 1 TO 10
  Row = Sheep.Height * Racer
  SET FONTSIZE 18
  SET FONT 3
  SET PENPOS Column, Row
  GPRINT Sheep$

  SET FONTSIZE 7
  SET PENPOS Column + 3, Row - 3
  GPRINT Racer
NEXT Racer
```

*(Listing continued on next page.)*

```

SET PENSIZ 2, 1
PLOT 225, 5; 225, Row
SET PENPOS 10, 200
SET FONTSIZ 12
GPRINT "They're at the gate...";
ASK PENPOS Message.Column, Message.Row
RETURN

Timer:
Oldtime$ = TIME$           ! Get original time
DO
  IF TIMES <> Oldtime$ THEN ! Has a second passed?
    Seconds = Seconds + 1   ! Increment counter
    Oldtime$ = TIME$       ! Reset the time
  ENDIF
  IF Seconds = 3 THEN EXIT  ! Been in loop 3 seconds yet?
  IF MOUSEB~ THEN EXIT     ! Mouse button pressed?
LOOP
Seconds = 0                ! Reset counter to 0
RETURN

Sheep.Mover:
Move = INT(RND(10)) + 1
Column = Sheep.Location(Move) ! This array element holds column
                                   ! number for sheep's 1st dot
Row = Sheep.Height * Move     ! Height is 16; count 16 dots
                                   ! from the top for each sheep

ERASE RECT Column, Row - Sheep.Height; Column + Sheep.Width, Row + 1
                                   ! This erases the "old" sheep
SET PENPOS Column + Sheep.Width, Row ! Move pen ahead 16 dots
                                   ! on this row
SET FONTSIZ 18                  ! Set up sheep's font size
SET FONT 3                     ! and font
GPRINT Sheep$                  ! Show that sheep

Sheep.Location( Move) = Sheep.Location( Move) + Sheep.Width
                                   ! Update the array element
                                   ! holding this sheep's position
SET FONTSIZ 7                  ! Set up font size for sheep's number
SET PENPOS Column + Sheep.Width + 3, Row - 3
                                   ! Same algorithm as in Field module
                                   ! for figuring where to show
                                   ! sheep's number
GPRINT Move                    ! Display the number
RETURN

```

*(Listing continued on next page.)*



## Racing.Messages:

```

Jump = Jump + 1                ! Keep track of all the moves
IF Sheep.Location(Move) > Sheep.Location(Leader) THEN Leader = Move

IF Jump < 11 THEN              ! Do this stuff if within first 10 moves
    SET PENPOS Message.Column, Message.Row
    SET FONTSIZE 16
    GPRINT "THEY'RE OFF!"
ELSE                            ! Do this stuff if beyond 10th move
    ERASE RECT 10, 180; 250, 220
    SET PENPOS 10, 200
    SET FONTSIZE 18
    GPRINT "In the lead: #"; Leader
ENDIF

```

```

RETURN

```

## Test.For.Winner:

```

Nose = Sheep.Location(Leader) + Sheep.Width
IF Nose > 226 THEN Winner = true
RETURN

```

## Declare.Winner:

```

Tail = Sheep.Location(Leader)    ! starting column coordinate
Head = Leader * Sheep.Height - 12 ! starting row coordinate
Foot = Leader * Sheep.Height + 1  ! ending row coordinate
                                   ! ("Nose" is ending column)

FOR Flash = 1 TO 10
    INVERT RECT Tail, Head; Nose, Foot
    FOR Stall = 1 TO 500          ! Here's the delay loop
        NEXT Stall
    NEXT Flash

ERASE RECT 10, 180; 250, 220      ! Lifted from Racing.Messages

SET PENPOS 10, 200
SET FONTSIZE 24                  ! Big type size
SET FONT 0                       ! Bold, thick letters
GPRINT Leader; " is the champ!"
RETURN

```

*(Listing continued on next page.)*

Go.Again:

```
SET VPOS 15
INPUT "Another race? (Y|N) "; Answer$
SELECT Answer$
    CASE "Y", "y"
        Repeat = true
    CASE ELSE
        Repeat = false
END SELECT
RETURN
```

Clear.Out:

```
CLEARWINDOW                ! Get rid of old race display
Jump = 0                    ! Clear dead counter
Winner = false              ! Clear dead flag
Leader = 0                  ! Clear old winner
FOR Clearout = 1 TO 10      ! Clear old sheep positions
    Sheep.Location(Clearout) = 0
NEXT Clearout
RETURN
```

End.Page:

```
CLEARWINDOW
SET PENPOS 65, 160
SET FONTSIZE 164
GPRINT Sheep$;              ! Need semicolon to stop sheep scroll

SET PENPOS 84, 110
SET FONTSIZE 24
SET FONT 0
GPRINT "Baa-ye.";           ! Need semicolon to stop sheep scroll
RETURN
```

Now, take a well-deserved break. Then call in your friends and neighbors to see *The Great American Sheep Race*. They'll love it!

### Some Possible Enhancements

There's all kinds of things you can do to change this program to make it more interesting. Here are some of the things I'd do if I had the time:

- Make the sheep hop across the track rather than just move.
- Add betting to the game so that an audience can participate.
- Announce when two sheep are "neck and neck" during the race.
- Every fifth move or so, announce who's in second and third place.
- Let the race continue until first, second and third place sheep cross the finish line.

Add your own ideas to this list. Your imagination is just as good as mine; let the old creative juices flow. The more you add your own ideas, the more the program reflects who you are.

### A Closing Comment

---

*The Great American Sheep Race* is a whimsical program, but the techniques you learned while writing it will serve you well no matter how serious and sophisticated your coding becomes. In this session, and indirectly throughout this whole book, you've been learning the rudiments of top-down programming, the method of writing code favored by most professional programmers and academicians. Top-down programming and the use of pseudocode can help you develop your programs clearly and logically.

This book has been a guided tour through some of the most important statements and structures in Macintosh BASIC. You didn't learn all of MacBASIC's keywords; there are too many of them to teach effectively in one volume. Now you can either get the second volume in this series (yes, there's another one!), experiment on your own to learn the rest of the keywords in the language, or both. Whichever route you choose, the important thing is to keep on coding.

Thanks for the use of the hall.

# APPENDIX



## Useful Tables

**Table A-1** Boolean Comparisons

Operator	Comparison	A to B*
=	equals	false
<	less than	true
>	greater than	false
<> or ≠	not equal	true
>= or ≥	not less than	false
<= or ≤	not greater than	true

- To get  $\neq$  press Option- $\neq$  (hold down the Option key while typing " $\neq$ ")
- To get  $\leq$  press Option- $\leq$  (hold down the Option key while typing " $\leq$ ")
- To get  $\geq$  press Option- $\geq$  (hold down the Option key while typing " $\geq$ ")

\*Assume that A holds 5 and B holds 6.

**Table A-2** Nonmatching Strings

First String	Second String	Reason For Nonmatch
"Bookstore "	"Bookstore"	First string has a space
"Edge Canyon"	"Edge canyon"	"C" and "c" don't match
"Rat.tails"	"Rat tails"	"." and " " don't match
" Summer"	"Summer"	Leading space in first string
Animal\$	Animals\$	Second string variable ends in s
Note.Pad\$	Note/Pad\$	"." and "/" don't match

**Table A-3** Rules of Precedence




Operator	Comment
( )	In case of nesting, innermost item is evaluated first
+ -	Unary operators (+6, -12)
* /	Standard multiplication and division
+ -	Standard addition and subtraction
= < >	Relational operators (not the assignment operator =)

**Table A-4** ASCII Character Chart (Geneva)

	56) 8	81) Q	106) j	131) É	156) ú	181) μ	206) Œ
32)	57) 9	82) R	107) k	132) Ñ	157) ù	182) ø	207) œ
33) !	58) :	83) S	108) l	133) Ò	158) û	183) Σ	208) -
34) " ' ;	59) ;	84) T	109) m	134) Û	159) ü	184) Π	209) -
35) #	60) < .	85) U	110) n	135) á	160) †	185) π	210) “
36) \$	61) =	86) V	111) o	136) â	161) °	186) ρ	211) ”
37) %	62) >	87) W	112) p	137) ã	162) ¢	187) σ	212) ‘
38) &	63) ?	88) X	113) q	138) ä	163) £	188) ρ	213) ’
39) ' @	64) @	89) Y	114) r	139) å	164) §	189) Ω	214) ÷
40) (	65) A	90) Z	115) s	140) å	165) ●	190) æ	215) ◇
41) )	66) B	91) [	116) t	141) ç	166) ¶	191) ø	216) ÿ
42) *	67) C	92) \	117) u	142) é	167) ß	192) ò	217) ➤
43) +	68) D	93) ]	118) v	143) è	168) ©	193) ï	
44) ,	69) E	94) ^	119) w	144) ê	169) ®	194) ñ	
45) -	70) F	95) _	120) x	145) ë	170) ™	195) ✓	
46)	71) G	96) `	121) y	146) í	171) ´	196) f	
47) /	72) H	97) a	122) z	147) ì	172) ¨	197) ≈	
48) 0	73) I	98) b	123) {	148) î	173) ≠	198) Δ	
49) 1	74) J	99) c	124)	149) ï	174) Æ	199) «	
50) 2	75) K	100) d	125) }	150) ñ	175) Ø	200) »	
51) 3	76) L	101) e	126) ~	151) ó	176) ∞	201) ...	
52) 4	77) M	102) f	127)	152) ò	177) ±	202)	
53) 5	78) N	103) g	128) Ä	153) ô	178) ≤	203) À	
54) 6	79) O	104) h	129) Å	154) õ	179) ≥	204) Ã	
55) 7	80) P	105) i	130) Ç	155) õ	180) ¥	205) Õ	



Table A-5 Available Fonts

Number	Name	Samples in Available Point Sizes
0	<b>System Font [Chicago]</b>	<b>12 point</b>
1	Application Font [Geneva]	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point
2	New York	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point, <b>36 point</b>
3	Geneva	9 point, 10 point, 12 point, 14 point, 18 point, 20 point, 24 point
4	Monaco	9 point, 12 point
5	<b>Venice</b>	<b>14 point</b>
6	<b>London</b>	<b>18 point</b>
7	<b>Athens</b>	<b>18 point</b>
8	<b>San Francisco</b>	<b>18 point</b>
9	Toronto	9 point, 12 point, 14 point, 18 point, 24 point
 [11]	 [Cairo]	 [18 point]
12	<i>Los Angeles</i>	12 point, 24 point

**Table A-6** Special Characters


































































































CHR\$(217) produces these special characters in the fonts and sizes shown:

	9 point	10 point	12 point	14 point	18 point	20 point	24 point
Font #2	♥	~	♫	♪	♥	~	🤖
Font #3	↔	📄	🐉	🏠	🐑	📺	🐯
Font #4	≡		🕯				
Font #5				🍷			
Font #6					🌸		
Font #7					🍄		
Font #8					🚗		
Font #9	♦		🍀	🍏	📦		🍃

Font #0 has three special characters in 12 point:

	CHR\$(17)	CHR\$(18)	CHR\$(20)
Font #0	⌘	✓	🍏

Table A-7 Font 11

CHR\$(33) = 	CHR\$(57) = 	CHR\$(81) = 	CHR\$(105) = 
CHR\$(34) = 	CHR\$(58) = 	CHR\$(82) = 	CHR\$(106) = 
CHR\$(35) = 	CHR\$(59) = 	CHR\$(83) = 	CHR\$(107) = 
CHR\$(36) = 	CHR\$(60) = 	CHR\$(84) = 	CHR\$(108) = 
CHR\$(37) = 	CHR\$(61) = 	CHR\$(85) = 	CHR\$(109) = 
CHR\$(38) = 	CHR\$(62) = 	CHR\$(86) = 	CHR\$(110) = 
CHR\$(39) = 	CHR\$(63) = 	CHR\$(87) = 	CHR\$(111) = 
CHR\$(40) = 	CHR\$(64) = 	CHR\$(88) = 	CHR\$(112) = 
CHR\$(41) = 	CHR\$(65) = 	CHR\$(89) = 	CHR\$(113) = 
CHR\$(42) = 	CHR\$(66) = 	CHR\$(90) = 	CHR\$(114) = 
CHR\$(43) = 	CHR\$(67) = 	CHR\$(91) = 	CHR\$(115) = 
CHR\$(44) = 	CHR\$(68) = 	CHR\$(92) = 	CHR\$(116) = 
CHR\$(45) = 	CHR\$(69) = 	CHR\$(93) = 	CHR\$(117) = 
CHR\$(46) = 	CHR\$(70) = 	CHR\$(94) = 	CHR\$(118) = 
CHR\$(47) = 	CHR\$(71) = 	CHR\$(95) = 	CHR\$(119) = 
CHR\$(48) = 	CHR\$(72) = 	CHR\$(96) = 	CHR\$(120) = 
CHR\$(49) = 	CHR\$(73) = 	CHR\$(97) = 	CHR\$(121) = 
CHR\$(50) = 	CHR\$(74) = 	CHR\$(98) = 	CHR\$(122) = 
CHR\$(51) = 	CHR\$(75) = 	CHR\$(99) = 	CHR\$(123) = 
CHR\$(52) = 	CHR\$(76) = 	CHR\$(100) = 	CHR\$(124) = 
CHR\$(53) = 	CHR\$(77) = 	CHR\$(101) = 	CHR\$(125) = 
CHR\$(54) = 	CHR\$(78) = 	CHR\$(102) = 	CHR\$(126) = 
CHR\$(55) = 	CHR\$(79) = 	CHR\$(103) = 	CHR\$(127) = 
CHR\$(56) = 	CHR\$(80) = 	CHR\$(104) = 	CHR\$(128) = 
			CHR\$(129) = 

# APPENDIX



## Commands, Special Characters, Keywords, and Statements

### Commands and Menu Items

---

**Alarm Clock**—desk accessory showing time and date. You can copy the time and date from it and paste them into a program.

**ASK set-option numeric-variable**—assign to *numeric-variable* the value of *set-option*.

**Backspace key**—delete selected material (same action as Clear); delete character just before insertion point.

**Calculator**—desk accessory used just like a simple four-function calculator. Results of all calculations are available for copying to the Clipboard and subsequent pasting to a BASIC program.

**Clear**—remove selected material entirely.

**Copy**—(⌘ C) copy selected material to Clipboard.

**Cut**—(⌘ X) remove selected material; move to Clipboard.

**Find**—(⌘ F) search for and highlight the next occurrence of the text named in the What to find dialog box.

**Go**—(⌘ G) execute the program in the active listing window.

**Halt**—(⌘ H) Stop the program whose output window is active.

**Key Caps**—desk accessory showing characters produced by various keys, including the “hidden” character set produced by holding down the Option key. Characters typed to Key Caps’ box can be cut or copied and later pasted into a BASIC program, either assigned to a string or displayed after the comment marker character.

**New**—(⌘ N) create a new listing window on the desktop with the title *Untitled*.

**Open Program file...**—(⌘ O) Bring into memory and make active a program whose name you choose from a list presented in a dialog box.

**Paste**—(⌘ V) insert material from the Clipboard into the active window at the insertion point.

**Replace**—(⌘ R) substitute specified string with a replacement string.

**Replace All**—substitute specified string with specified replacement string throughout a document.

**Run**—execute program in active listing window.

**Save Text**—(⌘ S) store a copy of the program in the active listing listing window on a disk under specified name.

**Select All**—(⌘ A) mark all material in the active window for some editing action.

**Undo**—(⌘ Z) cancel effect of most recent keypress or command in the active window.

**What to Find**—(⌘ W) specify parameters for FIND and REPLACE operations.



---

## Special Characters

---

- \$ character affixed to the end of a string variable name.
- ~ character affixed to the end of a boolean variable name.
- + addition operator.
- − subtraction operator.
- / division operator
- \* multiplication operator.
- = assignment operator; means “has the same value as.” Also relational operator meaning “equal to.”
- > relational operator meaning “greater than.”
- < relational operator meaning “less than.”
- ≠ relational operator meaning “not equal to”; type Option- = to get it. Same as <> or ><.
- ≥ relational operator meaning “not less than”; type Option-. to get it. Same as =< or >=.
- ≤ relational operator meaning “not greater than”; type Option-, to get it. Same as =< or <=.
- & string concatenation operator; used to combine two or more strings into one.
- ; character used at the end of a PRINT or GPRINT statement to prevent a carriage return/line feed.
- ! comment marker character; Macintosh BASIC ignores all text between it and the end of the line. You can use it as a program development and debugging tool to deactivate a line of code without removing the code from a listing.

---

## Keywords and Programming Statements

---

**ASC()**—numeric function taking a string argument (literal, variable, or expression) and returning the ASCII code for the first character in the argument's value.

**BTNWAIT**—interrupt program execution until the mouse button is pressed.

**CHR\$()**—string function taking a numeric argument (constant, variable, or expression) in the range 0 through 255 and returning the ASCII character or MacBASIC special character represented by the argument.

**CLEARWINDOW**—erase everything in the output window.

**DATA**—keyword marking beginning of data list. The only word that can precede DATA on an active program line is a label.

**DAT\$**—system string function taking no argument and returning the current date.

**DIM var(numexpr {[ ,numexpr]})**—set up an array called *var* with *numexpr* + 1 elements. An array can have additional dimensions up to the limits of memory; the number of elements reserved for each dimension must be stated and separated from its neighbors by a comma.

**DO\LOOP**—repeat indefinitely, and in order, the processing of all program lines falling between the keywords DO and LOOP.

**ELSE**—execute the statement between this keyword and the end of the line (or between it and the ENDIF keyword) if preceding condition evaluates as false.

**END PROGRAM**—immediately halt execution of program. Often used to protect program control from inadvertently falling into a subroutine code block.

**EXIT**—transfer program control out of loop structure to first statement following bottom of loop.

**FONT**—a *set-option* determining the current font (type style). The preset font in BASIC is 3, Geneva.

**FONTSIZE**—a *set-option* determining the height of a character. Each increment represents one point, or about  $\frac{1}{72}$  of an inch. Sizes below 6 are essentially unreadable. The preset size is 12.

**FOR *loop.var* = *start.val* TO *end.val* STEP *step.val* \NEXT *loop.var***—establish a looping structure wherein everything between the lines containing the keyword FOR and the keyword NEXT is repeated a specified number of times; variable *loop.var* increments in steps of *step.val* starting with the value *start.val* and ending when the value of *loop.var* exceeds that of *end.val*. Incrementing occurs when program flow reaches the NEXT line.

**FRAME *shape***—draw a solid line within the shape whose upper left point is described by coordinate set *left, top* and whose lower right point is described by coordinate set *right, bottom*. The coordinate sets are separated by a semicolon.

**GOSUB *label***—redirect program control to a subroutine whose code block begins with the single word *label*: and ends with the keyword RETURN.

**GPRINT**—display the specified character(s) in the predetermined FONT and FONTSIZE at the predetermined PENPOS.

**IF...THEN**—execute the statement between the keyword THEN and the end of the program line (or, if present, the keyword ELSE) if the condition between the keywords IF and THEN evaluates as true.

**IF...THEN \ELSE \ENDIF**—multiline IF...THEN...ELSE construct.

**INPUT**—get some information from the operator and store in the INPUT variable; display optional INPUT prompt.

**INT()**—a numeric function returning the lower whole number value of its argument (87.86 returns 87; -87.86 returns -88).

**INVERT *shape***—reverse the on-off state of each of the dots within the *shape* whose upper left point is described by the coordinate set *left, top* and whose lower right point is described by the coordinate set *right, bottom*. The coordinate sets are separated by a semicolon.

**LEFT\$()**—string function taking two arguments (a string and a numeric) and returning as many characters from the start of the first argument's string as is dictated by the second argument's value.

**LEN()**—numeric function taking one string argument and returning the number of characters in the argument.

**MID\$()**—string function taking two arguments (a string and a numeric) or three arguments (a string and two numerics) and returning as many characters from the first argument's string as is dictated by the other arguments. When one numeric is used, all characters from the position indicated by the numeric to the last character in the string are returned; when two numerics are used, as many characters as stipulated by the second numeric are returned, beginning with the character whose position is stipulated by the first numeric.

**MOUSEB**—system function returning a 1 when the mouse button is held down and a 0 when it is not down.

**MOUSEB~**—alternate, boolean form of the system function **MOUSEB**. This form allows you to use constructs like **IF MOUSEB~ THEN...** and **IF NOT MOUSEB~ THEN...** instead of **IF MOUSEB = 1 THEN...** and **IF MOUSEB = 0 THEN...**

**MOUSEH**—system function returning a number in the range  $\pm 32767$  indicating the horizontal position of the pointer relative to the left edge of the material in the Output window.

**MOUSEV**—system function returning a number in the range  $\pm 32767$  indicating the vertical position of the pointer relative to the top of the material in the Output window.

**NOT**—negate a boolean value; that is, produce *false* if the value is true, and vice-versa.

**OVAL**—BASIC graphics shape; a circular object inscribed within a rectangle whose upper left point is described by the coordinate set *left, top* and whose lower right point is described by the coordinate set *right, bottom*. The coordinate sets are separated by a semicolon.

**PAINT *shape***—fill with a specified pattern a specified shape whose upper left point is described by the coordinate set *left, top* and whose lower right point is described by the coordinate set *right, bottom*. If you don't specify a pattern, BASIC uses solid black.

**PATTERN**—a *set-option* in the range 0-37 determining what pattern BASIC will use with PAINT, or with PLOT and FRAME when the PENSIZE has been set sufficiently big. The preset pattern is 0, all black.

**PENPOS**—a *set-option* taking two parameters separated by a comma, which represent the horizontal and vertical coordinates on the graphics screen where the next graphics character will appear. The values of the parameters are between 0 and 32767. If the point specified is beyond the currently visible viewing area, you'll have to scroll the window to see it. The preset pen position is 0, 0.

**PENSIZE**—a *set-option* taking two parameters separated by a comma, which represent the height and width (in dots) of the pen point. Values are between 0 and 32767. The preset pen size is 1, 1.

**PLOT**—light up (or, more accurately, turn off) one or a series of dots. PLOT takes a minimum of one coordinate set.

**PRINT**—display some information in the output window.

**RANDOMIZE**—scatter the random number generator so that RND doesn't produce the same repeating series of numbers.

**READ *var***—retrieve the data list item immediately following the current position of the data marker and store in variable *var*; move the data marker past the retrieved item and to just before the following data item if another one exists.

**RECT**—BASIC graphics shape; a rectangular object whose upper left point is described by the coordinate set *left, top* and whose lower right point is described by the coordinate set *right, bottom*.

**RESTORE [*label*]**—position the data marker at the start of the first data list in the program. If the keyword RESTORE is followed by a label, then position the data marker at the start of the data list preceded by the matching label.

**RETURN**—make program control branch to the statement immediately following the one containing the keyword GOSUB that redirected program control to the subroutine in which RETURN appears.



**RIGHT\$()**—string function taking two arguments (a string and a numeric) and returning as many characters from the end of the first argument's string as is dictated by the second argument's value.

**RND()**—a numeric function returning a real number greater than zero and less than its argument.

**ROUNDRECT**—a graphics shape taking the ordinary parameters of a rectangle but ending in the syntactical phrase *WITH horizontal.roundness, vertical.roundness*. This phrase represents the degree of roundness of the shape's corners.

**SELECT *var* \CASE *literal* \[CASE ELSE \] END SELECT**—if *var* holds a value matching a literal that appears after any occurrence of the keyword CASE, execute the statements in that CASE's code block. If there's no other matching literal, execute the statements in the CASE ELSE block. A CASE's *literal* can be part of a range (*literal TO literal*), can have a relational operator before it (for example, *> literal*), or can be part of a series separated by commas.

**SET *set-option numeric-expression***—assign to *set-option* the value of the specified *numeric-expression*.

**TIMES\$**—system string function taking no argument and returning the current time.

**VPOS**—set option for moving the insertion point to a particular text row. VPOS works for PRINT text, but not for GPRINT text.

# APPENDIX



## Error Messages

Here's a list of all the error messages you're likely to encounter while using the keywords in this tutorial. If you get a message not on this list (which should happen only if your mistakes get really creative), look it up in the Errors appendix in the Macintosh BASIC Reference Manual.

**Can't find matching quote mark**—BASIC found one quote mark but can't find a matching second one. If you use one " you have to have a second one; the same is true for ', assuming it's not an apostrophe.

**Can't recognize statement**—Probably a spelling mistake, or you've embedded spurious spaces within a keyword.

**Can't recognize the keyword in SET or ASK**—The word you're using after SET or ASK isn't a set-option. Check your spelling.

**Can't recognize the rest of this line**—BASIC looks at a line of code and keeps reading characters until it thinks it has a whole legal statement. If there's anything on the line after the end of that statement (except for a colon or a comment), BASIC considers it trash.

**Couldn't find a Case that matched**—None of the numbers you use in your CASE lines matches the value of the variable after SELECT. There must be at least one matching CASE. To get around this restriction, use an empty CASE ELSE block:

```
N = 12
SELECT N
CASE 6
  PRINT "Hello"
CASE ELSE
  ! Just leave this empty—you don't even need a comment.
END SELECT
```

**Duplicate label**—You've used exactly the same label with different subroutines. BASIC can't tell which subroutine you want a GOSUB to jump to.

**ENDIF not found**—You've written a multiline IF construct without ending it with the keyword ENDIF. BASIC needs to see ENDIF so it knows which statements belong to the construct and which don't.

**Expected boolean expression**—You tried to use what looks to BASIC like a boolean construct but isn't. You'll get this error if you say something like IF A\$ THEN... because A\$ isn't a boolean expression.

**Expected close parenthesis**—right.

**Expected ELSE or carriage return**—You've typed too much text in an IF statement. For example, IF A = B THEN GOSUB C GOSUB D gets this error; the line should have ended after GOSUB C.

**Expected expression**—You didn't give BASIC a necessary expression. Probably you typed SELECT and then Return without providing a variable.

**Expected graphic object**—You used PAINT, FRAME, or INVERT but then didn't use RECT, OVAL or ROUNDRECT. Yet another example of leaving BASIC waiting for the proverbial other shoe.

**Expected literal value**—You used a variable where BASIC expected a literal. There aren't many situations in BASIC where this message comes up; usually it's in a CASE line of a SELECT structure.

**Expected a numeric**—BASIC wants to see nothing but a number or a numeric expression in many situations. You'll get this error; for example, if you give a string argument to a numeric expression like INT or RND.

**Expected operand**—You didn't give BASIC enough information. You'll get this error if you do something like type IF A = and then press Return.

**Expected a THEN keyword**—You started to write an IF statement but pressed RETURN before you finished it. Specifically, you wrote an IF statement with no THEN part.

**Illegal quantity**—BASIC finds a number in some situation where the number's size or sign doesn't make any sense, as in PRINT MID\$(Some\$, -6, 3).

**Inappropriate keyword**—You're using the wrong keyword for this situation.

**LOOP not found**—You started a DO loop but didn't finish it properly. Every DO must have a matching LOOP.

**LOOP without DO**—You finished a DO structure but didn't start it properly. Every LOOP must have a matching DO.

**Missing END SELECT statement**—You began a SELECT structure but didn't end it properly. Every SELECT must have a matching END SELECT.

**Missing keyword**—BASIC expected to see a keyword but didn't find one.

**Must end function with close parenthesis**—You can expect this one if you do a lot of nesting of functions; people tend to lose track of how many parentheses they've typed.

**Must have comma**—Many of BASIC's functions and graphics statements take lists of items, usually with the items separated by commas. Either you used a semicolon or some other characters or the punctuation is missing entirely.

**Must have semicolon**—You need to separate lists of coordinate pairs with semicolons.

**Must have variable after the NEXT**—Each FOR\NEXT construct ends with the keyword NEXT, immediately followed by the same variable name that appears right after FOR at the start of the structure.

**Must have a WITH keyword in ROUNDRECT**—This common error happens when you type a ROUNDRECT statement using the syntax for RECT. ROUNDRECT must always end with a WITH phrase.

**Negative subscripts not allowed**—Any expression that refers to an element of an array must be 0 or greater.

**NEXT without FOR**—This NEXT statement doesn't have a matching FOR; either FOR isn't there at all, or the variable name that appears after FOR doesn't match the one after NEXT.

**Not a statement**—Probably a spelling mistake.



**NOT operator requires boolean argument**—You used something other than a boolean expression after the keyword NOT. You can't say something like IF NOT Flag or IF NOT Flag\$, but you *can* say IF NOT FLAG`.

**Out of memory**—The program you've written has outgrown the computer's ability to hold it. Occasionally you'll get an error like this because you've written too much code; more often it means you've got a bug in your program. The bug probably has to do with some subroutine continuously calling itself in an infinite loop.

**RECT must follow ROUND**—You can never use the word ROUND by itself when BASIC can interpret it as being half of ROUNDRECT.

**RETURN without GOSUB**—You've allowed BASIC to wander into a subroutine without sending it there with a GOSUB statement. Protect your code from such mishaps by putting all your subroutines below an END PROGRAM statement.

**Subscript out-of-bounds**—BASIC has come upon an expression referencing an array element, but the value of the expression is greater than the value you used in the DIM statement. In effect, your program is referencing an array element that doesn't exist.

**Syntax error**—BASIC can't understand what you've typed. *Syntax error* is BASIC's way of saying "Uh...what?"

**Type mismatch**—Your code is mixing a type of value with a type of variable that doesn't match it. You can assign a boolean value only to a boolean variable, a string value to a string variable, and a numeric value to a numeric variable (although you can assign a number to a string, as long as the number is in quotes).

**Undefined label**—Your program tells BASIC to go to a subroutine that doesn't exist. Either the subroutine isn't in the code or you've spelled its name differently.

**Undimensioned array reference**—Your first reference to an array must be a DIM statement. Either you've forgotten to write this statement or it's in the wrong part of the code.

**Wrong number of subscripts**—The number of dimensions in your reference to an array doesn't match the number of dimensions in your DIM statement.

# APPENDIX



## Solutions to Bughouses

Here are the solutions to the Bughouse problems that appear at the end of most sessions. The corrections to the bugs are printed in **boldface** where possible. In many cases, there's more than one way to fix a bug. If your corrections are different from mine and the programs run properly, then you've solved the problem.

### Session 2

```
PRINT "I have always depended on the kindness of strangers."  
PRINT "But yah are, Blanche, yah are."
```

### Session 3

```
First.Num = 5  
Second.Num = 12  
Sum = First.Num + Second.Num  
PRINT Sum * 2  
INPUT "What do you think Sum equals? "; Sum.Guess  
PRINT Sum.Guess ; " was your guess."
```

### Session 4

```
DO  
  PRINT "What's Happening?"  
  Printout = Printout + 1  
  IF Printout = 10 THEN EXIT  
LOOP
```

**Session 5**

```
Words$ = "This is backloop # "  
Start = 10  
Finish = 1  
Increment = -1  
CLEARWINDOW  
FOR Number = Start TO Finish STEP Increment  
    PRINT Words$; Number  
NEXT Number
```

**Session 6**

```
Maximum = 200  
Change = 25  
DO  
    INVERT OVAL MOUSEH, MOUSEV; MOUSEH + Change, MOUSEV + Change  
    NewMouseH = Maximum - MOUSEH  
    NewMouseV = Maximum - MOUSEV  
    INVERT OVAL NewMouseH - Change, NewMouseV - Change; NewMouseH, NewMouseV  
    IF MOUSEB = 1 THEN CLEARWINDOW  
LOOP
```

**Session 7****RANDOMIZE**

```
FOR Numbers = 1 TO 10  
    GOSUB Get.Number  
NEXT Numbers  
END PROGRAM
```

Get.Number:

```
    PRINT "This random number is "; RND(100) + 1  
RETURN
```

**Session 8**

Code\$ = "Learning how to program in BASIC is a snap."

```
FOR Find.B = 1 TO LEN(Code$)  
    IF MID$(Code$, Find.B, 1) = "B" THEN EXIT  
NEXT Find.B  
  
Code$ = RIGHT$(Code$, LEN(Code$) - (Find.B - 1))  
BASIC$ = LEFT$(Code$, 5)  
  
PRINT Code$, ASC(RIGHT$(Code$, 1))  
PRINT BASIC$, ASC(Code$)
```

**Session 9**

DIM Bugs\$(6)

DO

Count = Count + 1

**READ** Bugs\$(Count)

Sentence\$ = Sentence\$ &amp; " " &amp; Bugs\$(Count)

IF Count = 6 THEN EXIT

LOOP

Sentence\$ = Sentence\$ &amp; "?"

PRINT Sentence\$

DATA When, do, we, do, more, graphics

**Session 10**

SET PATTERN 5

SET PENSIZE 5, 5

SET FONTSIZE 18

SET FONT 6

Left = 10

Top = 10

Right = 60

Bottom = 35

FOR Box = 1 TO 5

FRAME RECT Left, Top; Right, Bottom

SET PENPOS Left + 8, Top + 20

**GPRINT** "Box"

GOSUB Update:

NEXT Box

**END PROGRAM**

Update:

Top = Top + 35

Bottom = Bottom + 35

RETURN



**Session 11****DO****IF** MOUSEB<sup>~</sup> **THEN****CLEARWINDOW****SET** PENPOS RND(250), 0**ASK** PENPOS Column, Row**SELECT** Column**CASE** < 100**SET** PATTERN 23**PAINT** RECT 10, 10; 100, 200**CASE** 100 **TO** 200**SET** PATTERN 16**PAINT** OVAL 100, **140**; 200, 245 ! Whatever.**CASE** > 200**SET** PATTERN 4**PAINT** ROUNDRECT 100, 200; **275, 300** WITH 50, 50**END** **SELECT****ENDIF****LOOP**



# Glossary

**Active window**—window that is currently affected by commands.

**Algorithm**—a step-by-step procedure for solving a particular problem.

**Application font**—the preset font (typeface) for any Macintosh application. BASIC uses font 3, font size 12.

**Argument**—the expression enclosed within parentheses that a given function is to operate upon.

**Arithmetic operator**—one of four characters indicating addition, subtraction, division, or multiplication.

**Array**—variable structure in which a number of variables are referenced by the same name but by unique numeric subscripts.

**ASCII**—acronym for American Standard Code for Information Interchange, a coding system used by Macintosh BASIC to represent all text characters the machine is capable of reproducing.

**Assignment**—the process of giving a value to a variable.

**Assignment operator**—the character =. The variable to the left of = gets the value that appears to the right.

**Boolean**—referring to a true/false condition.

**Boolean variable**—variable whose name ends with the boolean character `~`. Boolean variables can hold one of only two values—true or false.

**Branching instruction**—an instruction to BASIC to transfer flow of control to another part of the program.

**Carriage return**—moves the insertion point to the left edge of the window; usually occurs with a line feed.

**Click**—to press and release the mouse button.

**Close box**—small box in upper left corner of active window which, when clicked, makes the window go away.

**Code**—instructions written in some computer language.

**Command**—an instruction to the computer that it carries out immediately. Commands appear in the various menus listed on the title bar. Contrast with *Statement*.

**Command key**—key you use to give BASIC a command from the keyboard using the  $\mathfrak{H}$  key rather than from a menu using the mouse.

**Comment**—a note to yourself (or to another programmer reading your code) that appears in a program line after the special symbol `!`. BASIC ignores comments; they're just for humans.

**Concatenation**—operation that combines two or more strings. The concatenation operator is the ampersand character, `&`.

**Conditional branch**—transfer of program flow from one part of the code to another, based on the outcome of some comparison.

**Constant**—that which never changes; generally applied to numbers to distinguish them from numeric variables.

**Control structure**—a group of BASIC statements bounded by keywords and operating in a reliable and consistent manner.

**Coordinate**—the point where a column and a row intersect.

**Coordinate set**—the two numeric values, first for column and second for row, describing a coordinate position. A comma separates the values.

**Counter**—a variable used to keep track of loop iterations.

**Data list**—one or more string literals or numeric constants following the keyword DATA, separated from each other by commas.

**Data pointer**—marker invisible to programmer and user that BASIC positions before the next available data item in a data list. READ uses it to know what item to take next. If there aren't any more data items, the marker remains at the end of the final data item in the program.

**Data type**—a specific kind of information, recognizable to BASIC as such by the character appearing (or *not* appearing) at the end of a variable name. This book covers three data types: string, numeric, and boolean.

**Debugging**—process of locating and removing errors (bugs) from a program.

**Delimiter**—ASCII character that separates items. For instance, commas are delimiters between items in data lists, open parentheses are delimiters between most function names and their arguments, and colons are generally delimiters between labels and subsequent programming statements.

**Desktop**—another word for your Macintosh's screen, used because your Macintosh always makes available to you so many of the items you usually find on a desk top (an alarm clock, note pad, etc.).

**Dialog box**—a box that appears on the screen anytime BASIC needs more information from you.

**Directory**—list of all BASIC programs on the disk. The directory appears when you use the Open Program file command.

**Double-click**—to press and release the mouse button twice.

**Drag**—to move a window by positioning the pointer on its title bar and then holding down the mouse button while rolling the mouse around.

**Editor**—Macintosh's built-in facilities for entering and modifying selected material.

**Element**—individual unit of an array, referenced by its numeric subscript.

**Expression**—any group of values meant to be taken as one value. Expressions can include constants, variables, functions, and operators.

**Flag**—a variable holding one of two values (usually true/false, 1/0, non-zero/zero, or positive number/negative number) that BASIC uses to determine which of two paths to follow.

**Flow of control**—order in which BASIC executes program lines.

**Function**—formula built into BASIC that manipulates numbers or strings in some highly specialized way. Most pocket calculators, for example, have a built-in square root function.

**Glass paper**—metaphor for the Macintosh's screen, so called because you use an electronic pen to write on it as if it were paper.

**Graphics area**—that area on the Macintosh capable of displaying and/or holding information regarding the placement of dots or pixels. Also, a specific area defined by coordinate sets.

**Humanize**—to make a program show its results or requests for information in a way that makes intellectual and aesthetic sense.

**I-beam**—shape of pointer when it appears in an area capable of being edited.

**Increment**—to increase the value of a variable by a specific value. You usually increment a variable repeatedly in a loop.

**Infinite loop**—endless repetition of one or more program lines.

**INPUT prompt**—optional string literal used in INPUT statements to let the operator know what information the computer needs.

**INPUT variable**—variable appearing at the end of an INPUT statement that accepts information from the operator.

**Integer**—number with no fractional or decimal part; a whole number.



**Interactive programming**—programming characterized by an ongoing exchange of information between the computer and the operator.

**Iteration**—one pass through a loop during which BASIC executes all statements within the body of the loop.

**Keyword**—word or phrase having a specific meaning to the computer.

**Label**—string of text characters beginning with a letter and ending with a colon, used to identify the beginning of a subroutine. Also used as a subroutine's name, with or without the colon, in the line that calls the subroutine.

**Line feed**—movement of the insertion point down one text line, usually with a carriage return.

**Listing window**—window holding a listing of a MacBASIC program.

**Loop**—one or more program lines whose action is repeated a number of times.

**Loop variable**—variable whose value changes on each pass through a loop.

**Menu bar**—bar running across the top of the Macintosh display, containing the names of all the available Macintosh menus. The menus hold all the Macintosh commands.

**Multidimensional array**—array with two or more lists referenced by the same array name but with more than one numeric in its subscript. There must be one numeric for each dimension; a comma separates each numeric from its neighbors.

**Nest**—to enclose one within another.

**Nested loops**—loops wholly enclosed within other loops.

**Null string**—a string containing no characters.

**Null string character**—theoretical character that returns an ASCII code of -1 when it appears as the argument to the function ASC. It doesn't really exist; BASIC just thinks it does.

**Numeric function**—function designed to manipulate a number or numeric expression. A numeric function usually takes one argument (enclosed in parentheses) and returns a single result.

**Numeric subscript**—numeric constant, variable, or expression appearing in parentheses immediately following an array name.

**Option key**—use this key to produce alternate characters. For example, Option-g produces the copyright symbol ©.

**Output window**—window displaying results of an executed MacBASIC program.

**Pass**—a single loop iteration.

**Pen**—metaphor describing that which changes dot states from off to on, or on to off. In effect, the pen lights up or darkens each dot.

**Pen point**—the tip of the graphics pen. The size of the pen point is determined by *set-option* PENSIZ.

**Pixel**—the smallest picture element; a single dot on the Macintosh graphics area, one of about 200,000.

**Point**—any coordinate location on the Macintosh display; also, when referring to font, about  $\frac{1}{72}$ " of type height.

**Pointer**—mouse's noseprint.

**Precedence**—the order in which BASIC performs calculations in a given expression or formula.

**Program**—set of coded instructions that makes a computer do something.

**Quoted string**—text enclosed in quotation marks.

**Real number**—number with a possible fractional part.

**Reference**—any naming of or calling attention to a variable or label by a program statement.

**Referencing line**—a line of code that contains a branching instruction (such as GOSUB). It's called a referencing line because it *refers to* a label that begins a subroutine in some other part of the program.

**Relational operator**—one of certain basic characters, used either singularly or in combination, whose function is to test the relationship between two values. These operators return a *true/false* result; some people call them boolean operators. They include =, <, >, ≠, ≥, ≤.

**Return key**—key on right-hand side of the keyboard; you press it to tell BASIC to act on what you typed.

**Runtime error**—error that BASIC can't find until you run a program.

**Scaled character**—a character whose size is such that it doesn't exist in any font stored on the BASIC disk, thus making it necessary for BASIC to create it based on some character that *does* exist; so called because BASIC must *scale* an existing character to match the specified size.

**Scroll bar**—vertical or horizontal bar, appearing to the left of and beneath an active window. You use it with the mouse to see hidden text.

**Scroll box**—small hollow box that appears in a scroll bar to let you know about hidden text.

**Select**—to mark text for action by some command or keystroke.

**Selected text**—text that is ready to have something done to it by an editing command.

**Set-option**—a special system variable. You give a parameter to a set-option through SET; you find out what the parameter is through ASK.

**Shell**—the Macintosh BASIC programming environment.

**Statement**—a keyword or keyword phrase that BASIC recognizes as an instruction. Contrast with *Command*.

**Status icon**—icon in upper-right corner of output window indicating “running” status of program: wavy lines ~ mean running, question mark ? means waiting for input, a hand ⎓ means program halted, a black square ■ means program ended, and a bug 🐛 means the Debugger is operating.

**String**—sequence of text characters.

**String literal**—anything appearing between quotes; same as quoted string. Any group of characters meant to be taken as they appear.

**String variable**—variable ending in the special character \$ (called dollar), capable of representing any series of text characters.

**Subroutine**—program block beginning with a label and ending with the keyword RETURN. BASIC executes a subroutine when it sees the keyword GOSUB immediately followed by the subroutine's label.

**Substring**—part of a string. BASIC finds a substring through one of the substring functions (LEFT\$, MID\$, RIGHT\$).

**Syntactic outline**—symbolic representation of the form of a BASIC construct or statement.

**Syntax error**—error in keyword spelling or punctuation; any error that occurs when you type a program line. BASIC finds these errors when you press Return.

**System function**—function that gives information about some changing state in the system, such as the current position of the mouse pointer.

**Text**—any collection of characters.

**Title bar**—horizontal bar running across the top of a window. The title bar holds the window's title and, in most cases, a close box.

**Typeface**—a particular style and size of type. BASIC has 11 different fonts in numerous type sizes.

**Type size**—the height of a particular character in points.

**Unary operator**—an operator that applies to a single value. For example, the minus sign in  $-7$  is said to be unary because it defines 7 as a minus number, as opposed to expressing an operation to be performed on two values (such as  $x - 7$ ).

**Variable**—character or group of characters representing a location in the computer's memory where BASIC stores a value.

**Variable structure**—manner in which a computer language stores values for variables.

# Index

## Symbols

- 
- |  |   |
|--|---|
| <p>&amp;<br/>\<br/>~<br/>!<br/>/<br/>\$<br/>=<br/>&gt;<br/>&gt;= or =&gt;<br/>&lt;<br/>&lt;= or =&lt;<br/>-<br/>.<br/>&lt;&gt; or ≠<br/>&lt;= or ≤<br/>-<br/>+<br/>?<br/>;</p> | <p>Ampersand, 164<br/>Backslash, 57<br/>Boolean symbol, 267<br/>Comments, 45–46<br/>Division, 32<br/>Dollar sign, 86<br/>Equal sign, 34, 62, 267<br/>Greater than, 62, 267<br/>Greater than or equal to, 62<br/>Less than, 62, 267<br/>Less than or equal to, 62<br/>Minus, 32<br/>Multiplication, 32<br/>Not equal, 267<br/>Not greater than, 267<br/>Not less than, 267<br/>Plus, 32<br/>Question mark, 42<br/>Semicolon, 38, 102</p> |
|--|---|
- 
- |  |   |
|--|---|
| <p>Active Clipboards, 22<br/>Active window, 19, 28<br/>Addition, 32, 51<br/>Alarm Clock, 178–80<br/>Algorithm, 151, 152<br/>    pseudocode as, 287<br/>    in random number program, 152<br/>American Standard Code for Information Interchange. <i>See</i> ASCII code<br/>Ampersand (&amp;), as concatenation operator, 164, 180, 181<br/>Apostrophes, problems with, 86<br/>Application font, 227, 233, 249<br/>Arguments, 130, 151<br/>    multiple, substring functions and, 158<br/>Arithmetic formulas, 32<br/>Arithmetic functions, in random number program, 152<br/>Arithmetic operators, 32–33, 50, 51<br/>Arrays, 185–203<br/>    <i>See also</i> Array variables<br/>    defined, 186, 214<br/>    multidimensional, 200–203, 214<br/>Array variables, 185–203<br/>    arithmetic with, 192–94<br/>    functions with, 195<br/>    referencing elements in, 191–92, 214<br/>    for strings, 195–99<br/>    structure of, 185–87<br/>ASC function, 170–72, 181</p> | <p>ASCII code, 167–68, 180<br/>    ASC function to display, 170–72, 181<br/>    assigning to strings, with CHR\$, 172–74, 181<br/>    charts, 168, 176<br/>    MacBASIC extensions, 174<br/>    null string character, 167, 172, 180<br/>    phantom, 172<br/>    for sheep character, 290<br/>    special font characters, 234–35<br/>    string comparisons and, 167–70<br/>ASK, with set-options, 247–48, 249<br/>Assignment, 34, 50, 86–91, 136–37, 204<br/>    data type requirements, 88<br/>    READ and DATA statements for, 204–13<br/>Assignment operator (=), 34, 50, 51<br/>    expressions with, 136–37</p> <p>Backslash symbol (\), as keyword separator, 57<br/>Backspace key, 28<br/>Backspacing, for removing text, 25<br/>    Undoing, 26–27<br/>Base Experiment program, 118–20, 124<br/>BASIC interpreter, 12<br/>BASIC shell, 13–14, 28<br/>Binary system, 12<br/>Boolean, defined, 265, 271</p> |
|--|---|



- Boolean operators, 265
  - hidden, 267
  - with SELECT constructs, 266
- Boolean symbol (^), 267, 271
- Boolean values, 264–70
- Boolean variables, 268–69, 271
  - in *Great American Sheep Race*, 302
- Booting, 7
- Branching, 61
  - conditional, 61, 65, 71
- Branching instruction, 143, 151
- BTWAIT statement, 110–11, 125
  - deactivating, 113
  - in Little Boxes program, 122–23
- Cairo, 227, 233
- Calculator, 192–94, 214
- Carriage return, 50
- CASE, 257–61, 271
- CASE ELSE block, 257–58
- CHR\$ function, 167, 172–74
  - defined, 181
  - for special font characters, 234–35
- Clear command, 25, 26, 29
  - quick-cleaning vs., 105
  - Undoing, 26–27
- CLEARWINDOW, 93, 95
- Clicking the mouse, 17, 28
- Clipboard
  - activating, 22
  - combining program blocks with, 309
  - with Edit functions, 20–25
  - quick-cleaning and, 105
  - role of, 20
  - viewing, 21, 22
- Close box, 28
  - in BASIC shell, 13
  - with Copy and Paste operations, 21
- Code, 12, 28
  - indenting for clarity, 54
  - planning, 90. *See also* Program planning
- Columns
  - in *Great American Sheep Race*, 288, 290, 291
  - pointer positioning in, 111–15, 125
- Combining PLOT statements, 104
- Command key options, 27, 28
- Commands, syntax requirements for, 35. *See also* specific command statements
- Commas
  - as data list delimiter, 208, 214
  - in data list items, 208
  - in SELECT\CASE\END SELECT constructs, 260–61
- Comments (!), 45–46, 50
  - temporary program lines, 113
- Compacting expressions, 136–37
- Comparing, IF... THEN statements for, 61–64
- Comparing strings, 165–70
- Concatenation, 164–65, 180
- Concatenation operator (&), 164, 180
- Conditional branching, 61, 65, 71
- Constants, 34, 50
- Control structure, defined, 73, 94
- Controlling blocks, in *Great American Sheep Race*, 308, 311
- Coordinates, 100, 124
- Coordinate set, 100, 124
- Copy, 21–22, 29
- Copyright symbol, 287
- Counters, 57–58, 71
  - clearing, 309, 310–11
  - DO\LOOP with, 57–58, 59–60
  - in FOR\NEXT loops, 74, 81
- Cut, 21–25, 29
  - for combining program modules, 303
- Data, information vs., 38
- DATA, 204, 215
  - in READ\DATA statements, 204–13
- Data lists, 204, 205–13, 214
- Data pointer, 205, 206, 214
  - RESTORE for moving, 207–11, 213
- Data types
  - Boolean values, 267–70
  - defined, 94
  - precautions against mixing, 88
  - for string variables, 88
- DATE\$ function, 177–80, 181
- Debugging, 12, 28
  - with Debugger, 55
  - importance of, 18
- Decimal numbers
  - converting to integers, 131–32, 134
  - RND function returns, 130, 152
- Decision making, advanced methods of, 253–73
- Delimiter, 208, 214
- Desk accessories
  - Alarm Clock, 178–80
  - Calculator, 192–94, 214
  - Key Caps, 174–76, 180
- Desktop, 50
- Dialog box, 48–49, 50
  - What to find, 196–98
- Dice Game program, 149–51, 269
- DIM statement, 187–90, 215
  - in *Great American Sheep Race*, 296, 309, 310
  - with multidimensional arrays, 200–203
- Directory, 69, 71
- Disks
  - combining program modules on, 303
  - inserting, 7
  - retrieving program modules from, 309
- Division symbol (/), as arithmetic operator, 32, 51
- DO\LOOP constructs, 54–60, 71
  - as control structure, 73
  - counters in, 57–58, 59–60
  - FOR\NEXT loops vs., 77
  - halting infinite loops, 56
  - INT function in, 132
  - iterating, 57–58
  - nesting in, 82–83
  - saving, 68
  - unlimited statements with, 68
  - use of EXIT with, 58–60
- Dollar sign (\$), as string variable terminator, 86, 95
- Dots (pixels), 99
  - line height and, 223–24
  - turning on and off, 101
- Double-click, 14, 28
- Dragging the pointer, 15, 28
- Editing, 18–29
  - Command key options for, 27, 28
  - commands, listed, 28. *See also* specific commands
  - defined, 18
  - selecting text for, 22–25
  - Undo command, 26–27
- Edit menu, 18
  - for retrieving program blocks, 309
- Editor, 18, 28
- Elements of an array, 186–87, 214
  - Element 0, 190
- ELSE. *See* IF... THEN... ELSE constructs
- END PROGRAM, 147, 151
- Equal sign (=), 267
  - as Boolean operator, 267
  - as relational operator, 61–62, 71
- ERASE RECT, in *Great American Sheep Race*, 297–99, 300, 301
- Error correction. *See* Debugging

- Error messages, 15–17
  - Assignment needs equals*, 15
  - cancel button precautions, 17
  - Dimension Too Big*, with multidimensional arrays, 203
  - with DO\LOOPs, 55
  - Ending quote not found*, 15
  - Extra garbage*, 15
  - with *Great American Sheep Race*, 310
  - with illegal variable names, 36
  - with null strings, 91
  - Out of Data*, with READ statements, 209
  - Out of Memory*, with multidimensional arrays, 203
  - Redimed array*, 310
  - RETURN without GOSUB, 148, 289
  - with Run command, 17
  - with subroutines, 148, 289
- Errors
  - when erasing, Undo for, 26–27
  - importance of, 18
  - runtime, 17
  - with Select and Paste commands, 20
  - structural, 17
- Exclamation point (!), as comment symbol, 45–46, 51
  - with temporary program lines, 113
- EXIT statement, 58–60, 71
  - in FOR\NEXT loops, 81
  - as Halt substitute, 58–60
  - in nested loops, 83–84
- Expressions
  - compacting, 136–37
  - defined, 71, 135, 151
  - nested, precedence and, 137–38
  - with relational operators, 136
- Fabulous Strings program, 163
- File menu, for retrieving program blocks, 309
- Find, 195, 214
- First Arrays program, 188–89
  - Search and Replace with, 196–98
- Flags
  - Booleans as, 269
  - clearing, 309, 310–11
  - defined, 269–271
  - in *Great American Sheep Race*, 302, 304, 305, 307, 309, 310–11
- Flashing, in *Great American Sheep Race*, 303
- Flow of control, 53, 71
  - conditional branching and, 61
  - defined, 71
  - EXIT and, 58–60
  - IF... THEN and, 58–60
  - loops change, 53
  - slowing with BTNWAIT, 110
- FONT, 226–29, 249
- Font characteristics program, 261–64
- Fonts
  - defined, 226, 249
  - displaying, 226–28
  - preset, 222
  - special font characters, 234–35
  - table of, 233
- FONTSIZE, 230–33, 249
- Formulas, 32
- FOR\NEXT loops, 74–85
  - defined, 95
  - DO\LOOP constructs vs., 77
  - EXIT in, 81
  - GOSUB statements with, 145–46
  - INPUTing variable values, 78–79
  - nesting in, 82–85
  - STEP and, 76–77
  - syntactic outline for, 74–75
  - variables in, 78–80
- FRAME, 108, 125
  - PAINT vs., 124
  - PLOT vs., 107
- FRAME OVAL, 124
  - SET PENSIZe with, 241
- FRAME RECT, 106–10
  - with mouse system functions, 118–21
  - set-options with, 240, 242
- Functions
  - See also* Numeric functions;
  - String functions
  - arguments of, 130, 151
  - defined, 111, 124
  - in random number program, 152
  - with string arrays, 198–99
- Geneva, as application font, 222, 227, 233
- Go, defined, 125
- GOSUB statement, 143–147, 151
- GOTO statement, 3
- GPRINT, 222–25, 249
- Grades program
  - multidimensional arrays in, 200–203
  - READ\DATA statements in, 207, 208
- Graphic pen position, nongraphic insertion point and, 305
- Graphic randomness, 139
- Graphics, 99–127
  - capability of Macintosh, 99
  - drawing a rectangle, 106–10
  - drawing lines, 102–4
  - PRINT and, 219–20
  - printing, 219–20, 222–25
  - ROUNDRECT, 242–47, 249
  - spacing between lines, 223–25
  - using loops for, 109–10
  - variables with, 107–10
- Graphics area, 99–100
  - bounds of, 112–16
  - in *Great American Sheep Race*, 275–317
- Clear.Out module, 309, 310–11
  - clearing flags and counters in, 309, 310–11
- combining modules, 303, 306–12
- controlling blocks, 308, 311
- Declare.Winner module, 307, 308
- detailed description, 279–84
- ending, 311
- end of race, 282–83, 291–92
- End.Page module, 283–84, 305–6, 307, 308
- Field module, 279–80, 289–93, 306, 307
- final testing, 312–16
- finish line and message, 291–92
- Go.Again module, 304–5, 306, 307, 308
- initializing routines in, 309–11
- messages during race, 281–82
- opening message, 279
- outcome, 277–78
- overview, 279
- possible enhancements to, 317
- pseudocode outline, 286–87
- race module code, 294–300
- race, 281
- Racing.Messages module, 300–302, 306, 307, 308
- rows in, 288, 290–291
- sheep for, 289–293
- Sheep.Mover module, 295–300, 306, 307, 308
- step 1, imagining the outcome, 277–78
- step 2, program specification, 278–85
- step 3, outlining specifications, 284–85
- step 4, writing code in modules, 287
- Test.For.Winner module, 302, 306, 307, 308
- Timer module, 293, 306, 307, 308
- Title.Page module, 288–89, 306, 307, 308
- writing code in modules, 287

- Greater than (>) symbol
  - as Boolean operator; 267
  - as relational operator; 62–64, 71
  - in SELECT constructs; 259
- Greater than or equal to (>=) or (= >), as relational operator; 62–64, 71
- Halt; 56, 71
- Horizontal scroll bar; 19
  - in BASIC shell; 13
  - with Copy and Paste operations; 21
- Humanizing a program; 42–44
- I-beam mouse pointer; 19, 28
  - in BASIC shell; 13
  - with Copy and Paste operations; 21
  - position functions; 111–23
  - role of; 13, 15
- Icons, locating; 7
- IF constructs, SELECT constructs vs.; 253, 256
- IF... THEN; 58–65, 71
  - for conditional branching; 61
  - with other keywords; 62–65
  - role of; 58, 61–64, 71
- IF... THEN... ELSE; 65–66, 71
- IF... THEN\ELSE\ENDIF; 253, 254–56, 271
- Incrementing; 57, 71
  - counters and; 57–58
- Index variables. *See* Loop variables
- Infinite loops; 56, 71
- Inflation program; 54–60
  - scroll bar use with; 66–68
- Information, data vs.; 38
- Initializing routines; 309–11
- Initializing variables; 310
- INPUT prompt; 41–45, 50
  - null string as response to; 90–91
  - syntax for; 44
- INPUT statements; 41–45, 50, 51
  - SET VPOS with; 304–305
  - with string variables; 87
- INPUT variables; 42, 50
  - with INT function; 132
- Insertion point; 21, 28
  - in BASIC shell; 13
- INT function; 129, 131–32, 152
- Integer Randomness program; 134–36
- Integers
  - converting decimal numbers to; 131–32, 134
  - defined; 131, 151
- Interactive programming; 41–42, 50, 51
- INVERT; 124, 125
- INVERT OVAL; 124
- INVERT RECT; 303, 304
- Iteration; 57–58, 71
  - in FOR\NEXT loops; 81–82
- Jump, in *Great American Sheep Race*; 281
- Keyboard, Boolean operator key
  - sequences; 267
- Key Caps; 174–76, 180
- Keywords; 15, 28
  - for drawing shapes; 107
  - typing; 50
- Labels
  - with DATA; 212
  - defined; 143, 151
  - with RESTORE; 221, 213, 215
  - use of colons with; 148, 151
- Languages; 12
- LEFT\$; 158–63
  - defined; 158, 181
- LEN function; 157–58, 181
- Less than (<) symbol
  - as Boolean operator; 267
  - as relational operator; 62–64, 71
  - with SELECT constructs; 259
- Less than or equal (<=) or (= <), as relational operator; 62–64
- Line feed; 51
- Lines, drawing; 102–4
- Listing window; 13–14, 28
  - activating; 14, 19
  - in BASIC shell; 13
  - editing in; 18–27
  - quick-cleaning; 104–5
  - scrolling through; 66–68
- Literal TO literal*; 258, 271
- Little Boxes program; 120–23
  - adding randomness to; 139
  - with BTNWAIT; 122–23
- Loop or index variables; 76–77, 94
  - iteration of; 81
- Loops; 53–72
  - counters in; 57–58
  - defined; 53, 71
  - DO\LOOP constructs; 54–60
  - with graphics programs; 109–10
  - nesting; 82–85
  - role of; 53
- LOOPS. *See* DO\LOOPS constructs
- Lowercase vs. uppercase letters
  - in ASCII code; 168
  - in Search and Replace operations; 197–98
- MacPaint
  - graphics clarification with; 225–26
  - in program planning; 277
- MacWrite, in program planning; 277
- Menu bar; 13, 28
  - moving pointer to; 15
- Messages, during *Great American Sheep Race*; 281–82. *See also* Error messages
- MID\$; 158–63, 181
- Minicalc program; 69–70
  - commented listing of; 45
- Minus sign (–), as arithmetic operator; 32, 51
- Modules
  - building program from; 306–12
  - as subroutines; 297, 308
  - writing code in; 287
- Mouse
  - booting MacBASIC with; 7, 14
  - BTNWAIT and; 110–11
  - clicking; 17, 28
  - double-clicking; 14
  - dragging; 15, 28
  - manipulating scroll box with; 66–68
  - pointer. *See* I-beam mouse pointer
  - pointer position functions and; 111–16
- MOUSEB; 116–17, 125
  - and loops, precautions with; 122–23
- MOUSEB~, as Boolean keyword; 270, 281
- MOUSEH, for current pointer column; 111–15, 125
  - out-of-bounds numbers and; 112–15
- MOUSEV, for current pointer row; 111, 115–16, 125
  - out-of-bounds numbers and; 116
- Multidimensional arrays; 200–203, 214
- Multiline IF... THEN\ELSE\ENDIF constructs; 253, 254–56, 271
- Multiplication symbol (.), as arithmetic operator; 32, 51

- Naming programs. *See* Program names
- Naming variables. *See* Variable names
- Nested expressions, precedence and, 137–38
- Nested loops, 82–85, 94
- Nesting, defined, 82, 94
- New command, 31, 50
- Not equal symbol (<> or ≠), 267, 273
- Not greater than symbol (<= or ≤), 267, 271
- Not less than symbol (>= or ≥), 267, 271
- NOT, to negate Boolean value, 269, 271
- Null string character, 167, 172, 180
- Null strings, 90–91, 94
- Numeric constants  
in data lists, 204, 212, 214  
expressions in place of, 136
- Numeric functions  
arrays with, 195, 198–99  
ASC, 170–72, 181  
defined, 151  
LEN, 157–58, 181  
role of, 129  
with string arrays, 198–99
- Numeric subscript, 186, 214
- Numeric variables, 31  
Counts as, 57  
quotes around, effect of, 88
- Open Data File command, 47
- Open Program command, 69, 71
- Operators  
arithmetic, 32–33, 50, 51  
assignment, 34, 50, 51  
relational, 61–64, 71  
with string functions, 198–99  
variable names prohibit, 35
- Option key, Boolean operators with, 267
- Out-of-bounds numbers, 112–15, 116
- Outlining program specifications, 281–282
- Output window, 17, 28  
scrolling through, 66–68  
size of, 99
- OVAL, 124, 125
- PAINT, 124, 125  
SET PATTERN with, 236–38, 249
- Pass, 76, 94
- Paste operations, 21–25, 29  
with Key Caps, 175, 176
- PATTERN options, 236–38, 249
- Pen point, 220, 249
- PENPOS, 221, 223, 249
- PENSIZE, 238–42, 249
- Phantom ASCII, 172
- Pixel (picture element), 99, 124
- PLOT statements, 100–104, 125  
combining, 104  
FRAME statements vs., 107  
line drawing with, 102–4  
SET PENPOS vs., 223  
SET PENSIZE with, 238–42
- Plus sign (+), as arithmetic operator, 32, 51
- Pointer. *See* I-beam mouse pointer
- Pointer position functions, 111–23
- Points, 222, 249  
defined, 230, 249  
sizes available, 230–33
- Precedence, 137, 151  
and nested expressions, 137–38
- PRINT statements, 15, 29  
font used with, 227  
graphics and, 219–20. *See also* GPRINT  
role of, 15, 29  
with semicolon, 38–41  
SET VPOS with, 304–305
- Program, defined, 12, 28
- Program comments. *See* Comments
- Programming control structure, defined, 94
- Programming environment, 11–29
- Program names, 49  
new versions, 198  
restrictions, 49  
typing, 50
- Program planning, 92, 275–87  
cash register program, 92  
combining modules, 303, 306–12  
controlling block, 308, 311  
final testing, 312–16  
imagining the outcome, 277–78  
initializing routines, 309–11  
program specification, 278–85  
pseudocode outline, 286–87  
writing code in modules, 287
- Programs  
comments in, 45, 46, 50  
entering, 14–17  
flow of control. *See* Flow of control  
humanizing, 43–44  
naming. *See* Program names  
role of, 11–12  
saving, 46–49  
temporary lines in, 113
- Program specification, 278–85  
detailed description, 279–84  
outlining, 284–85  
overview, 279
- Prompt messages, 43–45  
need for, 43  
null strings and, 90  
question mark, 42
- Pseudocode  
advantage of, 317  
for combining modules, 307–8  
in *Great American Sheep Race*, 280  
program outline as, 275, 286–87
- Question mark (?), 42–43
- Quotation marks  
apostrophes and, 86  
for data list items, 208, 212  
error messages about, 15  
numbers in, as string variables, 88  
as string delimiters, 86. *See also* Quoted strings  
string literals enclosed in, 39
- Quoted strings, 15, 28, 88, 89
- Random Boxes program, 139–42
- RANDOMIZE, 133–34, 152  
with INT function, 135  
with RND function, 134
- Random numbers, generating, 130–31, 133–34
- Random Ovals program, 142–43
- Random subroutines, 129–56
- Range, in SELECT constructs, 258–61, 271
- READ\DATA statements, 204–13
- Real number; defined, 130, 151
- Rectangles, 106–10, 125. *See also* FRAME RECT; INVERT RECT
- Redundancy, in variables, 87
- Referencing array elements, 191–92, 214
- Referencing line, defined, 148, 151
- Relational operators, 61–64, 71  
with CASE, in SELECT constructs, 259, 271  
expressions with, 136  
listed, 62  
for string comparisons, 165–70  
with string variables, 88–89
- Relational values, in BASIC processes, 264–65
- Replace, 195–96, 214
- Replace All, 195–96, 214
- RESTORE statement, 209–13, 215
- RETURN, as subroutine ending, 144, 151, 152
- Return key, 15, 28  
for creating null strings, 90

- 
- RETURN Without GOSUB, 148, 289
  - RIGHT\$, 158–63, 181
  - RND function, 129, 130–31, 141–42, 152
    - for numbers between 0 and 1, 143
  - ROUNDRECT, 243–47, 249
  - Rows
    - in *Great American Sheep Race*, 288, 290–91
    - pointer positioning in, 111, 115–16, 125
  - Run command, 17, 29
  - Runtime errors, 17, 28, 55
  
  - Save commands, 47–48, 51
    - See also* Saving a program
    - Save Binary, 48
    - Save a Copy in . . . , 47, 121–22, 125
    - Save Output, 47
    - Save Text, 47–49, 51
  - Saving a program
    - See also* Save commands
    - with new name, 198
    - with output window active, 68
    - with several versions, 121–22, 125
  - Scaled character, 232, 249
  - Scroll bars, 66–68, 71
  - Scroll box, 66–68, 71
  - Search menu, 195
  - Select, 28
    - manipulating selected text, 21–25
  - SELECT
    - IF constructs vs., 253, 256
    - to ignore upper/lower case difference, 305
  - Select All, 22, 29
  - SELECT\CASE\END SELECT, 253, 256–64, 274
    - with ranges, 258–61, 272
  - Semicolon (;)
    - with PLOT, for continuous lines, 102–4
    - in PRINT statements, 38–41, 51
  - Set-options, 247–48, 249
    - SET, 247, 249
    - SET FONT, 221, 226–29
    - SET FONTSIZE, 230–33
    - SET PATTERN, 236–38, 240
    - SET PENPOS, 221, 223, 249
    - SET PENSIZE, 238–42, 249
    - SET VPOS, 304–05
  - Shapes, multiple keywords for, 107.
    - See also* Graphics; specific shapes
  - Shell, 13–14, 28
  
  - Show Clipboard, 21
  - Size box, 13, 19
    - with Copy and Paste operations, 21
    - to ignore upper/lower case difference, 305
  - Spacing between lines, 223–25
  - STEP clause, 77
    - negative values and, 78–80
  - String arrays, 195–99
  - String concatenation, 164–65, 180, 181
  - String functions
    - arrays with, 195, 198–99
    - CHR\$, 167, 172–74
    - DATE\$, 177–80, 181
    - LEFT\$, 158–63, 181
    - MID\$, 158–63, 181
    - RIGHT\$, 158–63, 181
    - TIME\$, 177–80, 181
  - String literals, 39, 51
    - comparing, 169, 170
  - Strings, 157–83
    - comparing, 165–70
    - in data lists, 205, 208, 212–13
    - defined, 157
    - finding number of characters in, 157–58, 181
    - joining together, 164–65, 180
    - Key Caps for constructing, 174–76, 180
    - manipulating parts of, 158–63, 180
    - nonmatching, 89
    - numbers in quotes as, 88
    - quoted, 15, 28, 88, 89
    - substring functions, 158–63
  - String variables, 86–91
    - assigning values to, 86–91
    - comparing, 169, 170
    - data type for, 88
    - defined, 86, 94
    - dollar sign (\$) terminator, 86, 95
    - with INPUT statements, 87
    - matching and nonmatching, 89
    - naming, 84
    - null strings, 90–91, 94
    - quotation marks as delimiters in, 86, 87, 88
    - redundant, 87
    - with relational operators, 88–89
    - setting up, 86, 88
    - Sheep\$, 290
    - values of, 89
  - Structural errors, 17
  
  - Subroutines
    - Booleans in, 269
    - controlling block, *Great American Sheep Race*, 308
    - defined, 129, 144, 151
    - GOSUB statements in, 143–47
    - in *Great American Sheep Race*, 291
    - modules as, 297, 308
    - random, 129–36
    - Sheep.Mover module, 296–97
    - testing, 288–89
    - Title.Page, 288–89
  - Substring functions, 158–63
  - Substrings, defined, 180
  - Subtraction, 32, 51
  - Syntactic outline, defined, 74, 94
  - Syntax, 12
    - errors, 28
    - variable name restrictions, 35
  - System functions
    - defined, 111, 124
    - MOUSEB, 116–17, 125
    - MOUSEH, 111–15, 125
    - MOUSEV, 111, 115–16, 125
    - names for, 117
  - System variables, set-options as, 247–48, 249
  
  - Text
    - clearing, 25–26, 29
    - defined, 14, 28
    - editing. *See* Editing
    - entering, 15–17
  - TIME\$ function, 177–80, 181
  - Timer module, in *Great American Sheep Race*, 29
  - Title bar 19, 28
    - in BASIC shell, 13
    - to ignore case difference, 305
  - Top-down programming, 275, 287
    - advantages of, 317
  - True or false answers. *See* Boolean operators
  - Two-dimensional arrays, 200–203
  - Typefaces. *See* Fonts
  
  - Unary operator, defined, 151
  - Undefined variables, 268
  - Undo, 26–27, 29
  - Untitled, 14
  - Uppercase vs. lowercase letters
    - in ASCII code, 168
    - conventions, 50
    - in Search and Replace operations, 197–98
    - in variable names, 36



Variable names, 34, 35–37  
     system function names vs., 117

Variable names table, 36

Variables, 31–51

- in arithmetic formulas, 32–33
- arrays, 185–203. *See also* Arrays
- assigning, 34, 86–91, 204–13
- data types defined, 94
- defined, 31, 51
- with graphic objects, 107–9
- incrementing, 57
- initializing, 310
- INPUTing, 41–44
- located in memory, 36
- loop or index, 76–77, 94
- in looping structures, 78–80
- naming. *See* Variable names
- redundant, 87
- string. *See* String variables
- typing, 50

Variable structures, 185–87, 214

Vertical scroll bar, 19

- with Copy and Paste operations,  
     21
- to ignore case difference, 305

VPOS, 304–305

What to Find dialog box, 196–98

Windows

- active, 19, 28
- clearing, 93, 95
- coordinates and, 100
- listing. *See* Listing window
- opening, 7
- output, 17, 28, 66–68, 99
- scrolling through, 66–68
- untitled, 14, 16



# Introduction to Macintosh™ BASIC

Enter the programming environment of Apple's Macintosh BASIC complete with many built-in programming tools, unique commands and coding instructions, the powerful Macintosh Editor, and special accessories for developing, testing, and debugging your programs. The twelve sessions in this book are full of keyboard and hands-on-mouse activities that provide the framework for exploring Macintosh BASIC. The author reveals the innovative elegance and simplicity of this new BASIC language that will enable you to write programs never before possible in BASIC. To experience Macintosh BASIC is to discover a fresh, exciting way to program even if you're a veteran BASIC programmer.

Learn both the concepts and the practice of solid programming including how and where to use variables, loops, subroutines, random numbers, and arrays. Develop and refine your programming skills through the exercises and pop quizzes that appear in each session. Quickly master the special world of Macintosh graphics and animation. And find creative techniques and hints to improve your programming style as you create your own Macintosh BASIC masterpieces.

Appendices include the Macintosh BASIC character set; a command summary; a list of error messages; and an ASCII chart for quick, easy reference. ■



## About the Author

Scot Kamins has been affiliated with Apple Computer Inc. since its earliest days.

Most recently, Scot worked as a key member of the Macintosh BASIC Language Design Team. He wrote Apple's *Macintosh BASIC Programmer's Reference Manual* and the *Applesoft Programmer's Reference*

*Manual* for the IIc and IIe. Scot also created the manual *Hands-on BASIC Interactive*

*Tutorial* for Eduware. He is cofounder of the San Francisco Apple Core and a senior writer with Technology Translated, Inc., of San Francisco. Scot has also published software and taught Computer Science.



**HAYDEN BOOK COMPANY**

a division of Hayden Publishing Company, Inc.

Hasbrouck Heights, New Jersey

ISBN 0-8104-6550-7