

# Problem Solving with Data Structures: A Multimedia Approach

Mark Guzdial and Barbara Ericson  
College of Computing/  
Georgia Institute of Technology

**ALPHA VERSION OF TEXT**



July 22, 2009



Copyright held by Mark Guzdial and Barbara Ericson, 2009.

Dedicated to our families. We are grateful for their support.

# Contents

<b>Contents</b>	<b>iii</b>
<b>List of Program Examples</b>	<b>vii</b>
<b>List of Figures</b>	<b>xii</b>
<b>I Introduction to Java: Object-Oriented Programming for Modeling a World</b>	<b>7</b>
<b>1 Objects for Modeling a World</b>	<b>9</b>
1.1 Making Representations of a World . . . . .	10
1.2 Why Java? . . . . .	17
<b>2 Introduction to Java</b>	<b>21</b>
2.1 What's Java about? . . . . .	21
2.2 Basic (Syntax) Rules of Java . . . . .	23
2.3 Using Java to Model the World . . . . .	33
2.4 Manipulating Pictures in Java . . . . .	47
2.5 Exploring Sound in Java . . . . .	53
2.6 Exploring Music in Java . . . . .	54
<b>3 Methods in Java: Manipulating Pictures</b>	<b>61</b>
3.1 Reviewing Java Basics . . . . .	61
3.2 Java is about Classes and Methods . . . . .	66
3.3 Methods that return something: Compositing images . . . . .	74
3.4 Creating classes that do something . . . . .	84
<b>4 Objects as Agents: Manipulating Turtles</b>	<b>89</b>
4.1 Turtles: An Early Computational Object . . . . .	89
4.2 Drawing with Turtles . . . . .	90
4.3 Creating animations with turtles and frames . . . . .	98
4.4 Making a Slow Moving Turtle with sleep and exceptions . . . . .	103
<b>5 Arrays: A Static Data Structure for Sounds</b>	<b>109</b>
5.1 Manipulating Sampled Sounds . . . . .	109

5.2	Inserting and Deleting in an Array . . . . .	115
5.3	How Slow Does It Get? . . . . .	119
<b>II Introducing Linked Lists</b>		<b>123</b>
<b>6</b>	<b>Structuring Music using Linked Lists</b>	<b>125</b>
6.1	JMusic and Imports . . . . .	125
6.2	Making a Simple Song Object . . . . .	130
6.3	Making a Song Something to Explore as a Linked List . . .	132
<b>7</b>	<b>Structuring Images using Linked Lists</b>	<b>163</b>
7.1	Simple arrays of pictures . . . . .	164
7.2	Listing the Pictures, Left-to-Right . . . . .	164
7.3	Listing the Pictures, Layering . . . . .	170
7.4	Reversing a List . . . . .	179
7.5	Animation . . . . .	180
7.6	Lists with Two Kinds of Elements . . . . .	183
<b>III Trees: Hierarchical Structures for Media</b>		<b>201</b>
<b>8</b>	<b>Trees of Images</b>	<b>203</b>
8.1	Representing scenes with trees . . . . .	203
8.2	Our First Scene Graph: Attack of the Killer Wolveries . . . . .	204
8.3	The Classes in the SceneGraph . . . . .	206
8.4	Building a scene graph . . . . .	210
8.5	Implementing the Scene Graph . . . . .	218
8.6	Exercises . . . . .	233
<b>9</b>	<b>Lists and Trees for Structuring Sounds</b>	<b>237</b>
9.1	Composing with Sampled Sounds and Linked Lists: Recursive Traversals . . . . .	237
9.2	Using Trees to Structure Sampled Sounds . . . . .	254
<b>10</b>	<b>Generalizing Lists and Trees</b>	<b>275</b>
10.1	Refactoring a General Linked List Node Class . . . . .	275
10.2	Making a New Kind of List . . . . .	285
10.3	The Uses and Characteristics of Arrays, Lists, and Trees . .	287
10.4	Binary Search Trees: Trees that are fast to search . . . . .	294
<b>11</b>	<b>Abstract Data Types: Separating the Meaning from the Implementation</b>	<b>309</b>
11.1	Introducing Stacks . . . . .	309
11.2	Introducing Queues . . . . .	323
11.3	Using an ArrayList . . . . .	335
11.4	Using a map ADT . . . . .	337

<b>12 Circular Linked Lists and Graphs: Lists and Trees That Loop</b>	<b>343</b>
12.1 Making Sprite Animation with Circular Linked Lists . . . . .	343
12.2 Generalizing a Circular Linked List . . . . .	349
12.3 Graphs: Trees with Loops . . . . .	351
<b>13 User Interface Structures</b>	<b>359</b>
13.1 A Toolkit for Building User Interfaces . . . . .	359
13.2 Rendering of User Interfaces . . . . .	362
13.3 A Cavalcade of Swing Components . . . . .	374
13.4 Creating an Interactive User Interface . . . . .	380
13.5 Running from the Command Line . . . . .	397
<b>IV Simulations: Problem Solving with Data Structures</b>	<b>403</b>
<b>14 Using an Existing Simulation Package</b>	<b>405</b>
14.1 Introducing Simulations . . . . .	405
14.2 Overview of Greenfoot . . . . .	408
14.3 Greenfoot Basics . . . . .	413
14.4 Creating new classes . . . . .	415
14.5 Breakout . . . . .	420
<b>15 Introducing UML and Continuous Simulations</b>	<b>433</b>
15.1 Our First Model and Simulation: Wolves and Deer . . . . .	433
15.2 Modeling in Objects . . . . .	436
15.3 Implementing the Simulation Class . . . . .	440
15.4 Implementing a Wolf . . . . .	444
15.5 Implementing Deer . . . . .	449
15.6 Implementing AgentNode . . . . .	450
15.7 Extending the Simulation . . . . .	450
<b>16 Abstracting Simulations: Creating a Simulation Package</b>	<b>459</b>
16.1 Creating a Generalized Simulation Package . . . . .	460
16.2 Re-Making the Wolves and Deer with our Simulation Package	468
16.3 Making a Disease Propagation Simulation . . . . .	477
16.4 Making a Political Influence Simulation . . . . .	485
16.5 Walking through the Simulation Package . . . . .	491
16.6 Finally! Making Wildebeests and Villagers . . . . .	495
<b>17 Discrete Event Simulation</b>	<b>515</b>
17.1 Describing a Marketplace . . . . .	515
17.2 Differences between Continuous and Discrete Event Simu- lations . . . . .	516
17.3 Different Kinds of Random . . . . .	518
17.4 Ordering Events by Time . . . . .	529

17.5 Implementing a Discrete Event Simulation . . . . .	538
17.6 The Final Word: The Thin Line between Structure and Behavior . . . . .	556
<b>A MIDI Instrument names in JMusic</b>	<b>561</b>
<b>Bibliography</b>	<b>565</b>
<b>Index</b>	<b>567</b>

# List of Program Examples

Example Program: An Example Program . . . . .	4
Example Program: Person class, starting place . . . . .	34
Example Program: Person, with a private name and public accessor and modifier methods . . . . .	35
Example Program: Person, with constructors . . . . .	37
Example Program: toString method for Person . . . . .	39
Example Program: Student class, initial version . . . . .	40
Example Program: Class Student, with constructors and toString method . . . . .	41
Example Program: main() method for Person . . . . .	44
Example Program: greet method for Person . . . . .	45
Example Program: greet method for Student . . . . .	45
Example Program: Method to double the red in a Picture . . . . .	50
Example Program: Method to flip an image . . . . .	52
Example Program: Method to increase red in Picture . . . . .	65
Example Program: decreaseRed in a Picture . . . . .	69
Example Program: decreaseRed with an input . . . . .	72
Example Program: Method to compose this picture into a target . . . . .	74
Example Program: Method to scale the Picture by a factor . . . . .	78
Example Program: Methods for general chromakey and blueScreen . . . . .	82
Example Program: A public static void main in a class . . . . .	84
Example Program: Creating a hundred turtles . . . . .	93
Example Program: Making a picture with dropped pictures . . . . .	97
Example Program: An animation generated by a Turtle . . . . .	99
Example Program: SlowWalkingTurtle . . . . .	104
Example Program: Increase the volume of a sound by a factor . . . . .	110
Example Program: Reversing a sound . . . . .	111
Example Program: Create an audio collage . . . . .	112
Example Program: Append one sound with another . . . . .	113
Example Program: Mix in part of one sound with another . . . . .	114
Example Program: Scale a sound up or down in frequency . . . . .	114
Example Program: Inserting into the middle of sounds . . . . .	117
Example Program: Deleting from the middle of a sound . . . . .	118
Example Program: <i>Amazing Grace</i> as a Song Object . . . . .	130
Example Program: SongNode class . . . . .	133

Example Program: A SongPhrase class . . . . .	136
Example Program: Computed Phrases . . . . .	146
Example Program: 10 random notes SongPhrase . . . . .	148
Example Program: 10 slightly less random notes . . . . .	149
Example Program: Repeating and weaving methods . . . . .	149
Example Program: RepeatNextInserting . . . . .	152
Example Program: SongPart class . . . . .	155
Example Program: Song class—root of a tree-like music structure . .	156
Example Program: MySong class with a main metho0d . . . . .	158
Example Program: Elements of a scene in position order . . . . .	165
Example Program: Methods to remove and insert elements in a list	169
Example Program: LayeredSceneElements . . . . .	171
Example Program: Reverse a list . . . . .	179
Example Program: Create a simple animation of a dog running . . .	181
Example Program: Abstract method drawWith in abstract class SceneElement	185
Example Program: SceneElement . . . . .	186
Example Program: SceneElementPositioned . . . . .	190
Example Program: SceneElementLayererd . . . . .	191
Example Program: MultiElementScene . . . . .	192
Example Program: Modified drawFromMeOn in SceneElement . . .	194
Example Program: The drawing part of DrawableNode . . . . .	207
Example Program: Start of WolfAttackMovie class . . . . .	210
Example Program: Setting up the movie in WolfAttackMovie . . . . .	211
Example Program: Rest of setUp for WolfAttackMovie . . . . .	213
Example Program: Rendering just the first scene in WolfAttackMovie	214
Example Program: renderAnimation in WolfAttackMovie . . . . .	215
Example Program: DrawableNode . . . . .	220
Example Program: PictNode . . . . .	223
Example Program: BlueScreenNode . . . . .	224
Example Program: Branch . . . . .	225
Example Program: HBranch . . . . .	228
Example Program: VBranch . . . . .	230
Example Program: MoveBranch . . . . .	231
Example Program: SoundElement . . . . .	238
Example Program: SoundListText: Constructing a SoundElement	
list . . . . .	246
Example Program: RepeatNext for SoundElement . . . . .	248
Example Program: Weave for SoundElement . . . . .	248
Example Program: copyNode for SoundElement . . . . .	249
Example Program: De-gonging the list . . . . .	250
Example Program: replace for SoundElement . . . . .	251
Example Program: SoundTreeExample . . . . .	255
Example Program: CollectableNode . . . . .	258
Example Program: SoundNode . . . . .	260
Example Program: SoundBranch . . . . .	262

Example Program: ScaleBranch . . . . .	264
Example Program: LLNode, a generalized linked list node class . . . . .	276
Example Program: DrawableNode, with linked list code factored out . . . . .	279
Example Program: CollectableNode, with linked list code factored out . . . . .	282
Example Program: StudentNode class . . . . .	286
Example Program: TreeNode, a simple binary tree . . . . .	296
Example Program: insertInOrder for a binary search tree . . . . .	299
Example Program: find, for a binary search tree . . . . .	301
Example Program: traverse, a binary tree inorder . . . . .	303
Example Program: addFirst and addLast, treating a tree as a list . . . . .	305
Example Program: A Stack Abstract Data Type (Interface) . . . . .	312
Example Program: Stack implemented with a linked list . . . . .	314
Example Program: Stack implemented with an array - declaration, fields, and constructor . . . . .	317
Example Program: Stack implemented with an array—methods . . . . .	317
Example Program: Reverse a list—repeated . . . . .	321
Example Program: Reverse with a stack . . . . .	321
Example Program: A Queue Abstract Data Type (Interface) . . . . .	324
Example Program: A Queue implemented as a linked list . . . . .	325
Example Program: A Queue implemented as an array . . . . .	327
Example Program: Defining an Abstract Queue Class . . . . .	331
Example Program: The Revised ArrayQueue Class . . . . .	332
Example Program: The Revised LinkedListQueue Class . . . . .	333
Example Program: The ArrayListStack Class . . . . .	335
Example Program: Oil paint Picture method . . . . .	337
Example Program: Find the most common color method in Pixel . . . . .	338
Example Program: WalkingKid . . . . .	346
Example Program: CharacterNode, a class for representing characters in sprite animations . . . . .	350
Example Program: A Simple GUItree class . . . . .	362
Example Program: Slightly more complex GUItree class . . . . .	364
Example Program: A Flowed GUItree . . . . .	365
Example Program: A BorderLayout GUItree . . . . .	369
Example Program: A BorderLayout GUItree . . . . .	372
Example Program: An interactive GUItree . . . . .	383
Example Program: Start of RhythmTool . . . . .	385
Example Program: Starting the RhythmTool Window and Building the Filename Field . . . . .	386
Example Program: Creating the Count Field in the RhythmTool . . . . .	388
Example Program: RhythmTool's Buttons . . . . .	389
Example Program: Imports and fields for the PictureTool class . . . . .	391
Example Program: Constructor for the PictureTool class . . . . .	392
Example Program: Start of the setUpMenu method for the PictureTool class . . . . .	393

Example Program: Handling the file menu items in the PictureTool class . . . . .	394
Example Program: Creating the tools menu in the PictureTool class . . . . .	394
Example Program: Handling the tools menu items in the PictureTool class . . . . .	395
Example Program: The rest of the PictureTool class . . . . .	396
Example Program: Test program for String[] args . . . . .	397
Example Program: The main method in PictureTool . . . . .	397
Example Program: The act method in the Wombat class . . . . .	409
Example Program: The turnRadom method in the Wombat class . . . . .	411
Example Program: The act method with a random turn . . . . .	412
Example Program: The Start of the WombatWorld Class . . . . .	413
Example Program: The rest of WombatWorld . . . . .	414
Example Program: Methods to randomly place leaves, wombats, and walls . . . . .	416
Example Program: The modified Wombat constructor . . . . .	418
Example Program: The changed act method in Wombat . . . . .	418
Example Program: The changed canMove method in Wombat . . . . .	419
Example Program: The beginning of the BreakWorld class . . . . .	421
Example Program: The method that sets up the breakout game . . . . .	423
Example Program: The method that creates and sets the background image . . . . .	423
Example Program: The setUpBricks method . . . . .	424
Example Program: The Brick Class . . . . .	425
Example Program: The beginning of the Ball class . . . . .	426
Example Program: The Ball act method . . . . .	428
Example Program: The newBall method in World . . . . .	429
Example Program: The World method that checks if the user has won . . . . .	430
Example Program: WDSimulation's setUp() method . . . . .	468
Example Program: WDSimulation's toString() method . . . . .	470
Example Program: The start of DeerAgent including the init method . . . . .	470
Example Program: DeerAgent's die() method . . . . .	472
Example Program: DeerAgent's act() method . . . . .	472
Example Program: DeerAgent's constructors . . . . .	473
Example Program: WolfAgent's fields, getWolves, and init method . . . . .	474
Example Program: WolfAgent's act() method . . . . .	475
Example Program: DiseaseSimulation's setUp method . . . . .	477
Example Program: WDSimulation's toString method . . . . .	479
Example Program: PersonAgent's init method . . . . .	479
Example Program: PersonAgent's act method . . . . .	480
Example Program: PersonAgent's infect method . . . . .	481
Example Program: PersonAgent's getNumInfected method . . . . .	481
Example Program: PoliticalSimulation's setUp method . . . . .	486
Example Program: PoliticalSimulation's lineForFile and endStep methods . . . . .	487
Example Program: PoliticalAgent's declaration and fields . . . . .	488

Example Program: PoliticalAgent's init method . . . . .	488
Example Program: PoliticalAgent's setParty method . . . . .	489
Example Program: PoliticalAgent's act method . . . . .	489
Example Program: BirdSimulation's class declaration, fields, and setUp method . . . . .	497
Example Program: BirdSimulation's endStep method . . . . .	498
Example Program: Changing Simulation's run() method for a time step input to act() . . . . .	499
Example Program: Changing Agent to make time step inputs op- tional . . . . .	499
Example Program: BirdAgent's class declaration, fields, and init method 500	
Example Program: BirdAgent's act method . . . . .	501
Example Program: EggAgent's declaration and fields . . . . .	503
Example Program: EggAgent's init method . . . . .	503
Example Program: EggAgent's act method . . . . .	504
Example Program: Generating random numbers from a uniform dis- tribution . . . . .	520
Example Program: Generate a histogram . . . . .	521
Example Program: Generate normal random variables . . . . .	525
Example Program: Generating a specific normal random distribution	528
Example Program: Event Queue Exerciser . . . . .	531
Example Program: EventQueue (start) . . . . .	532
Example Program: Insertion Sort for EventQueue . . . . .	535
Example Program: Inserting into a sorted order (EventQueue) . . .	536
Example Program: SimEvent (just the fields) . . . . .	545
Example Program: Truck . . . . .	546
Example Program: Distributor . . . . .	550
Example Program: FactorySimulation . . . . .	554

# List of Figures

1.1	Wildebeests in <i>The Lion King</i> . . . . .	9
1.2	Parisian villagers in <i>The Hunchback of Notre Dame</i> . . . . .	10
1.3	Katie's list of treasure hunt clues . . . . .	12
1.4	An organization chart . . . . .	13
1.5	A map of a town . . . . .	14
1.6	Bedroom note refers to the kitchen . . . . .	15
1.7	Memory mailboxes for bedroom note . . . . .	15
1.8	Treasure hunt trail . . . . .	16
1.9	Modified treasure hunt trail . . . . .	16
1.10	Open the DrJava Preferences by clicking on Edit and then Preferences . . . . .	19
1.11	Adding <code>java-source</code> , the <code>jMusic</code> libraries, and the jars in <code>java-source</code> to the classpath in DrJava . . . . .	20
1.12	Parts of the DrJava window . . . . .	20
2.1	UML class diagram of movie types and movie showings . . . . .	22
2.2	Showing the API for the <code>String</code> class . . . . .	27
2.3	Showing memory for an <code>int</code> variable and a <code>String</code> variable . . . . .	27
2.4	A UML class diagram for <code>Person</code> and <code>Student</code> . . . . .	40
2.5	A modified UML class diagram for <code>Person</code> and <code>Student</code> . . . . .	43
2.6	Showing a picture . . . . .	49
2.7	Doubling the amount of red in a picture . . . . .	50
2.8	Doubling the amount of red using our <code>doubleRed</code> method on <code>bigben.jpg</code> . . . . .	52
2.9	Flipping our guy—original (left) and flipped (right) . . . . .	54
2.10	Just two notes . . . . .	56
2.11	Structure of a score in terms of <code>JMusic</code> objects . . . . .	57
3.1	Structure of the <code>Picture</code> class defined in <code>Picture.java</code> . . . . .	67
3.2	Part of the JavaDoc page for the <code>Pixel</code> class . . . . .	73
3.3	Composing a guy into the jungle . . . . .	76
3.4	Mini-collage created with <code>scale</code> and <code>compose</code> . . . . .	80
3.5	Using the <code>explore</code> method to see the sizes of the guy and the jungle . . . . .	81
3.6	Chromakeying the guy onto a blank screen using <code>blueScreen</code> and different thresholds for <code>chromakey</code> . . . . .	82

3.7	Run the main method from DrJava . . . . .	85
4.1	Starting a Turtle in a new World . . . . .	91
4.2	A drawing made by a turtle . . . . .	92
4.3	What you get with a hundred turtles starting from the same point, each turning a random amount from 0 to 360, then moving forward the same amount . . . . .	95
4.4	Dropping the monster character . . . . .	96
4.5	Dropping the monster character after a turtle rotation . . . . .	96
4.6	An iterated turtle drop of a monster . . . . .	97
4.7	Making a more complex pattern of dropped pictures . . . . .	98
6.1	Playing all the notes in a score . . . . .	126
6.2	Keys, MIDI notes (numbers), and frequencies . . . . .	127
6.3	Viewing a multipart score . . . . .	129
6.4	JMusic documentation for the class Phrase . . . . .	129
6.5	Trying the Amazing Grace song object . . . . .	132
6.6	A single SongNode instance . . . . .	139
6.7	First score generated from ordered linked list: first node, second node, then first node connected to second node . . . . .	140
6.8	A score made of up of three nodes, by inserting the riff (middle window) into the old list (top window), resulting in the bottom window . . . . .	141
6.9	Javadoc for the class SongNode . . . . .	146
6.10	Repeating a node 5 times . . . . .	151
6.11	The new phrase (riff2) . . . . .	152
6.12	Weaving a new node among the old . . . . .	152
6.13	Multi-part song using our classes . . . . .	159
7.1	Array of pictures composed onto a background . . . . .	164
7.2	Elements to be used in our scenes . . . . .	165
7.3	Our first scene . . . . .	168
7.4	Our second scene . . . . .	168
7.5	Removing the doggy from the scene . . . . .	170
7.6	Inserting the doggy into the scene . . . . .	171
7.7	First rendering of the layered sene . . . . .	176
7.8	A doubly-linked list . . . . .	177
7.9	Second rendering of the layered sene . . . . .	178
7.10	A few frames from the AnimatedPositionedScene . . . . .	184
7.11	The abstract class SceneElement, in terms of what it knows and can do . . . . .	186
7.12	The abstract class SceneElement and its two subclasses . . . . .	191
7.13	A scene rendered from a linked list with different kinds of scene elements . . . . .	193
7.14	Same multi-element scene with pen traced . . . . .	195

8.1	A simple scene graph . . . . .	205
8.2	A more sophisticated scene graph based on Java 3-D . . . . .	205
8.3	The nasty wolvies sneak up on the unsuspecting village in the forest . . . . .	206
8.4	Then, our hero appears! . . . . .	206
8.5	And the nasty wolvies scamper away . . . . .	207
8.6	Mapping the elements of the scene onto the scene graph . . . . .	207
8.7	Stripping away the graphics—the scene graph is a tree . . . . .	208
8.8	UML class diagram of classes used in our scene graph . . . . .	209
8.9	The forest branch created in setUp—arrows point to children . . . . .	212
8.10	Reserving more memory for Interactions using DrJava’s Preferences pane . . . . .	217
8.11	The implementation of the scene graph overlaid on the tree abstraction . . . . .	219
8.12	The actual implementation of the scene graph . . . . .	219
9.1	The initial SoundElement list . . . . .	244
9.2	As we start executing playFromMeOn() . . . . .	245
9.3	Calling e2.collect() . . . . .	245
9.4	Finally, we can return a sound . . . . .	246
9.5	Ending e2.collect() . . . . .	246
9.6	Starting out with e1.replace(croak,clap) . . . . .	252
9.7	Checking the first node . . . . .	252
9.8	Replacing from e2 on . . . . .	253
9.9	Finally, replace on node e3 . . . . .	253
9.10	Our first sampled sound tree . . . . .	254
9.11	The core classes in the CollectableNode class hierarchy . . . . .	258
9.12	Extending the class hierarchy with ScaleBranch . . . . .	259
9.13	Starting out with tree.root().collect() . . . . .	267
9.14	Asking the root’s children to collect() . . . . .	267
9.15	Asking the first SoundNode to collect() . . . . .	268
9.16	Asking the next SoundNode to collect() . . . . .	268
9.17	Collecting from the next of the SoundBranch . . . . .	269
9.18	Collecting from the last SoundBranch . . . . .	270
10.1	An example organization chart . . . . .	290
10.2	An equation represented as a tree . . . . .	291
10.3	A tree of meanings . . . . .	291
10.4	A sample sentence diagram . . . . .	292
10.5	A query for a collection of sentence trees . . . . .	293
10.6	A user interface is a tree . . . . .	294
10.7	Simple binary tree . . . . .	294
10.8	More complex binary tree . . . . .	295
10.9	Tree formed by the names example . . . . .	300
10.10	An unbalanced form of the last binary search tree . . . . .	302
10.11	Rotating the right branch off “Sam” . . . . .	303

10.12	An equation tree for different kinds of traversals . . . . .	304
11.1	A pile of plates—only put on the top, never remove from the bottom . . . . .	310
11.2	Later items are at the head (top) of stack . . . . .	310
11.3	New items are inserted at the top (head) of the stack . . . . .	311
11.4	Items are removed from the top (head) of a stack . . . . .	311
11.5	An empty stack as an array . . . . .	319
11.6	After pushing Matt onto the stack-as-an-array . . . . .	320
11.7	After pushing Katie and Jenny, then popping Jenny . . . . .	320
11.8	A basic queue . . . . .	323
11.9	Elements are removed from the top or head of the queue . . . . .	323
11.10	New elements are pushed onto the tail of the queue . . . . .	324
11.11	When the queue-as-array starts out, head and tail are both zero . . . . .	330
11.12	Pushing Matt onto the queue moves up the head to the next empty cell . . . . .	330
11.13	Pushing Katie on moves the head further right . . . . .	330
11.14	Popping Matt moves the tail up to Katie . . . . .	331
11.15	The Inheritance Tree for the Interface List . . . . .	334
11.16	Manipulating a picture to look like it was painted . . . . .	340
12.1	Scenes from Nintendo's Super Mario Brothers . . . . .	344
12.2	Three images to be used in a sprite animation . . . . .	345
12.3	A sequence of images arranged to give the appearance of walking . . . . .	345
12.4	A circular linked list of images . . . . .	346
12.5	Frames of the walking kid . . . . .	346
12.6	A partial circular linked list . . . . .	349
12.7	A map as a graph . . . . .	352
12.8	Apply weights to a graph—distances on a map . . . . .	353
12.9	Traversing a graph to create a spanning tree . . . . .	354
12.10	Choosing the cheapest path out of Six Flags . . . . .	355
12.11	Going to College Park . . . . .	355
12.12	Backtracking to avoid re-visiting Six Flags . . . . .	356
12.13	Adding Dunwoody, the obviously cheaper path . . . . .	356
12.14	Finishing up in Charlotte . . . . .	357
13.1	Examples of Swing components: JFrame, JPanel, and JSplitPane . . . . .	360
13.2	A simple GUI . . . . .	364
13.3	A slightly more complex GUI, with two buttons . . . . .	365
13.4	Our GUItree, using a Flowed Layout Manager . . . . .	366
13.5	Diagram of components of GUI tree . . . . .	367
13.6	Resizing the Flowed GUItree . . . . .	367
13.7	How a BorderLayout GUI is structured . . . . .	369
13.8	A BorderLayout GUItree . . . . .	370
13.9	Resizing the BorderLayout GUItree . . . . .	371
13.10	Example of a GridBag layout . . . . .	371

13.11	Our GUItree rendered by the BorderLayout . . . . .	373
13.12	Resizing the BorderLayout GUItree . . . . .	373
13.13	Example of use of JScrollPane . . . . .	375
13.14	Example of a JTabbedPane . . . . .	375
13.15	Example of JToolBar . . . . .	375
13.16	An example of JOptionPane . . . . .	376
13.17	An example of JInternalFrame . . . . .	376
13.18	An example of a JComboBox . . . . .	376
13.19	An example of a JSlider . . . . .	377
13.20	An example of a JProgressBar . . . . .	377
13.21	An example of JColorChooser . . . . .	378
13.22	An example of JFileChooser . . . . .	378
13.23	An example of a JTextField . . . . .	379
13.24	An example of a JPasswordField . . . . .	379
13.25	An example of a JTextArea . . . . .	379
13.26	A tool for generating rhythms . . . . .	385
13.27	A tool for manipulating pictures . . . . .	391
13.28	Executing PictureTool from the command line with a file name as input . . . . .	398
14.1	The ants scenario in Greenfoot . . . . .	409
14.2	The balloons scenario . . . . .	410
14.3	The lunerlander scenario . . . . .	410
14.4	Creating wombats and leaves in the wombat scenario . . . . .	411
14.5	Switching to the documentation from the editor . . . . .	412
14.6	Creating a subclass of Actor . . . . .	415
14.7	Creating a Wall class . . . . .	416
14.8	A new WombatWorld . . . . .	419
14.9	The breakout game display . . . . .	421
15.1	An execution of our wolves and deer simulation . . . . .	434
15.2	A diagram of the relationships in the Wolves and Deer simulation	435
15.3	A UML class diagram for the wolves and deer simulation . . . .	438
15.4	One UML class . . . . .	438
15.5	A Reference Relationship . . . . .	439
15.6	A Gen-Spec (Generalization-Specialization) relationship . . . .	440
15.7	The structure of the wolves linked list . . . . .	442
16.1	Sample of disease propagation simulation . . . . .	464
16.2	A Political Influence Simulation . . . . .	464
16.3	UML diagram of the base Simulation Package . . . . .	465
16.4	UML class diagram for Wolves and Deer with the Simulation Package . . . . .	469
16.5	UML Class Diagram of Disease Propagation Simulation . . . .	478
16.6	A graph of infection in the large world . . . . .	483
16.7	A graph of infection in a smaller world . . . . .	484

16.8 UML class diagram of political simulation . . . . . 485

16.9 Mapping from agent (turtle) positions on the left to character  
positions on the right . . . . . 496

16.10 Frames from the Egg-Bird Movie . . . . . 496

16.11 The individual images for the bird characters . . . . . 501

16.12 The various egg images . . . . . 504

17.1 A distribution where there is only an increase . . . . . 519

17.2 A histogram of 5000 random values from a uniform distribution 525

17.3 Our histogram of 5000 normal random values . . . . . 527

17.4 Histogram drawn from a normal distribution to our specifications 530

17.5 An example of a min-heap . . . . . 538

17.6 Adding a new value to the min-heap . . . . . 538

17.7 After switching the 10 and 3 values . . . . . 539

17.8 The final min-heap after adding a new value . . . . . 539

17.9 Screenshot of factory simulation running . . . . . 540

17.10 UML class diagram of the discrete event simulation package . . 543

17.11 UML class diagram with factory simulation classes . . . . . 544



# Preface

In 1961, the MIT Sloan School hosted a symposium on Computers and the World of the Future. The speakers were a whos-who of computer science pioneers, including Grace Hopper, John McCarthy, J.C.R. Licklider, and Alan Perlis, the first ACM Turing Awardee. Perlis' lecture[3] argued that everyone should take computer science as part of a modern liberal education. He listed the topics of a computer science class for everyone, which included:

*Representation.* The data organization in each problem may have a natural representation for one phase of a computation and a different one for another. Transformations between representations must always balance in the students mind the ease of processing versus the work involved in transformation...

*Simulation.* It is important for the student to realize that a process possibly not even associated with, and operating external to, a computer may to varying degrees of approximation be imitated on a computer...

This book addresses these two points. Most data structures texts deal with *representation*, but few also deal with *simulation*. The combination provides the opportunity to think about the reason for the representations. This book addresses data structures as tools that you can use to solve problems that arise in modeling a world and executing (simulating) the resultant model. We cover the standard data structure topics (e.g., arrays, linked lists, stacks, queues, trees, maps, and graphs) but in the context of modeling. The execution of the model results in a simulation which often generates an animation. The last section of the book challenges the reader to think about structure and behavior more abstractly, and how they interrelate. The following are some of the strengths of this book:

- The same problem is addressed in multiple ways, increasing in sophistication and power with each iteration. For example, Chapter 6 considers the problem of supporting music composition on a computer. First, we consider arrays of notes, then linked lists of MIDI (music) note segments. We develop several different kinds of linked

lists, that have increasing flexibility for music composition and expression. In Chapter 7, we consider the problem of laying out images to define a scene, and work through several iterations of linked lists and finally develop scene graphs, a common data structure in computer animation. That's the first kind of tree that students see in this book. Simulations are introduced through the use of an existing simulation package in Chapter 14, then a simple predator-prey simulation is created based on our Turtle class in Chapter 15, and a complete general continuous simulation package is developed in Chapter 16, and finally discrete event simulations are covered in Chapter 17.

- The same data structure is seen in multiple contexts. Chapter 6 introduces linked lists for MIDI notes, Chapter 7 introduces linked lists for images, Chapter 9 uses linked list (with recursive traversals) for sampled sounds, and then a generalized form of linked lists (for any kind of data) is developed in Chapter 10. A similar progression is used with trees, using images in Chapter 8, sounds in Chapter 9, and creating the generalized structure in Chapter 10. In this way, we believe that students develop an abstract notion of data structures.
- Since we are using a movie context our data can be images and sounds instead of just numbers. For example, we create linked lists of musical phrases and trees of images and sounds. The advantage to this approach is that students find the exercises much more motivating than the traditional abstract ones. This approach also allows for open-ended assignments that encourage the students to be creative. In our use of this book at Georgia Tech 70% of the students reported that working with media made the class more interesting and 66% of the students reported spending extra time on a project to make the outcome look “cool”[6].

## How to use this book

Many teachers will want to tailor how they use this book.

- If students already know Java, then skip sections 2.1 - 2.3 and section 3.1. You may think that you can also skip some of chapter 4 but the concepts introduced here are used in later animations and simulations.
- Chapter 5 can be skipped if your students are already familiar with traversing and modifying one dimensional arrays.
- Chapters 6, 7, 8, and 9 introduce linked lists and trees using different media. These might be used stand-alone (e.g., using these chapters within the context of an existing data structures class) or used selectively (e.g., just Chapter 6 for music, or just Chapters 7 and 8 for

pictures). Chapter 10 generalizes on the previous linked lists and trees, so it does presume that some of the previous chapters have been visited.

- Chapter 13 is about user interfaces. It can be skipped, but you might want to at least cover section 13.2 which explains how user interfaces relate to trees. And section 13.4 uses a stack to allow users to undo previous operations.
- Part 4 of the book (chapters 14-17) is about simulations. This is where the data structures get put together to create new kinds of solutions. This part could be used as a final project in a more traditional data structures class.

## Expectations of the Reader

The presumption is that the reader has had some previous programming experience. We expect that the reader can build programs that use variables, iteration, and conditionals and that the reader can assemble programs using subprograms (functions/methods) that pass input via arguments. The reader should also be familiar with arrays. We don't care what language the readers previous experience is in.

We use DrJava in the examples in this text. *It is not necessary to use Dr- Java to use this book!* The advantages of DrJava are that it has a simple interface and a powerful interactions pane. The interactions pane allows us to manipulate objects without writing new classes or methods for each exploration. Rapid iteration allows students to explore and learn more quickly than they might if each exploration required a new Java class and/or method.

## Typographical notations

Examples of Java code look like this: `x = x + 1`. Longer examples look like this:

```
public static void main(String[] args) {  
    System.out.println("Hello, World!");  
}
```

When showing something that the user types in with DrJava's response, it will have a similar font and style, but the user's typing will appear after a prompt (`>`):

```
> int a = 5;  
> a + 7  
12
```

User interface components of DrJava will be specified using a smallcaps font, like the EDIT menu item and the LOAD button.

There are several special kinds of sidebars that you'll find in the book.

### Utility Program

#### Utility #1: An Example Utility

Utility programs are new pieces that we use to construct our models—not necessarily to be studied for themselves, but offered as something interesting to study and expand upon. They appear like this:

```

2  public class Greeter {
      public static void main(String[] argv) {
          // show the string "Hello World" on the console
4     System.out.println("Hello World");
      }
6  }

```

### Program Example #0

#### Example Java Code: An Example Program

A program creates a model of interest to us.

```

import jm.music.data.*;
2  import jm.JMC;
import jm.util.*;
4  import jm.music.tools.*;

6  public class Dot03 {
      public static void main(String[] args) {
8         Note n = new Note(JMC.C4, JMC.QUARTER.NOTE);
          Phrase phr = new Phrase(0.0);
10
          phr.addNote(n);
12         Mod.repeat(phr, 15);

14         Phrase phr2 = new Phrase(0.0);
          Note r = new Note(JMC.REST, JMC.EIGHTH.NOTE);
16         phr2.addNote(r);
          Note n2 = new Note(JMC.E4, JMC.EIGHTH.NOTE);
18         phr2.addNote(n2);
          Note r2 = new Note(JMC.REST, JMC.QUARTER.NOTE);
20         phr2.addNote(r2);
          Mod.repeat(phr2, 7);
22
          Part p = new Part();
24         p.addPhrase(phr);
          p.addPhrase(phr2);
26
          View.show(p);

```

28        }  
          }

**Computer Science Idea: An Example Idea**

Powerful computer science concepts appear like this.

**A Problem and Its Solution: The Problem that We're Solving**

We use data structures to solve problems that arise when we model a world. In these side bars, we explicitly identify the problem and its solution.

**Common Bug: An Example**

Common things that can cause your program to fail appear like this.

**Debugging Tip: An Example**

If there's a good way to keep those bugs from creeping into your programs in the first place, they're highlighted here.

**Making It Work Tip: An Example**

Best practices or techniques that really help are highlighted like this.

**Acknowledgements**

Our sincere thanks go out to the following:

- The National Science Foundation who gave us the initial grants that started the Media Computation project;

- Robert “Corky” Cartwright and the whole DrJava development team at Rice University;
- Andrew Sorensen and Andrew Brown, the developers of JMusic;
- Monica Sweat, Colin Potts, David Smith, and especially Jay Summet, who used this text at Georgia Tech and gave us feedback on it;
- Finally but most importantly, Matthew, Katherine, and Jennifer Guzdial, who allowed themselves to be photographed and recorded for Mommy and Daddy’s media project.

## **Part I**

# **Introduction to Java: Object-Oriented Programming for Modeling a World**



# 1 Objects for Modeling a World

In the 1994 Disney animated movie *The Lion King*, there is a scene when wildebeests charge over the ridge and stampede the lion king, Mufasa (Figure 1.1<sup>1</sup>). Later, in the 1996 Disney animated movie *The Hunchback of Notre Dame*, Parisian villagers mill about, with a decidedly different look than the rest of the characters (see the bottom of Figure 1.2<sup>2</sup>). These are actually related scenes. The wildebeests' stampede was one of the rare times that Disney broke away from their traditional hand-drawn cel animation. The wildebeests were not drawn by hand at all—rather, they were *modeled* and then brought to life in a *simulation*.

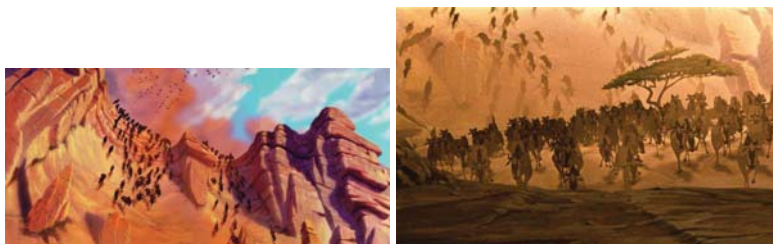


Figure 1.1: Wildebeests in *The Lion King*

A model is a detailed description of structure and behavior. The model of the wildebeests for *The Lion King* described what wildebeests looked like, how they moved, and what they did during a stampede. The villagers' model described what they did when milling about and how they reacted as a group to something noteworthy, like the entrance of Quasimodo. A simulation is an execution of the model—simply let the wildebeests start responding to one another and to the obstacles on the ridge, according to the behavior defined in their model. Then, in a sense, simply “film” the screen.

This is a different process than what Pixar used when it created *Toy Story*. There was a model for Woody, which described how Woody looked

---

<sup>1</sup>Images copyright ©1994, 1995 The Walt Disney Company.

<sup>2</sup>Images copyright ©1995, 1996 The Walt Disney Company.



Figure 1.2: Parisian villagers in *The Hunchback of Notre Dame*

and what parts of him moved together when he smiled or walked. But *Toy Story* wasn't a simulation. The movements and character responses of *Toy Story* were carefully scripted. In the wildebeest or villagers simulations, each character is simply following a set of rules, usually with some random element (e.g., Should the wildebeest move left or right when coming up against the rock? When should the villagers shuffle or look right?) If you run a simulation a second time, depending on the model and the random variables you used, you may get a different result than you did the first time.

This book is about understanding these situations. The driving questions of this book are ***“How did the wildebeests stampede over the ridge?”*** and ***“How did the villagers move and wave?”***. The process of answering those questions will require us to cover a lot of important computer science concepts, like how to choose different kinds of *data structures* to model different kinds of structures, and how to define behavior and even combine structure and behavior in a single model. We will also develop a powerful set of tools and concepts that will help us understand how to use modelling and simulation to answer important questions in history, science, or business.

## 1.1 Making Representations of a World

When we create models we construct a representation of the world. Think about our job as being the job of an artist—specifically, let's consider a painter. A painter creates a model of the world using paints, brushes, and a canvas. We will use the programming language *Java* to create model worlds.

Is there more than one way to model the world? Can you imagine two different paintings, perhaps *radically* different paintings, of the same thing? Part of what we have to do is to pick the software structures that

best represent the structure and behavior that we want to model. Making those choices is solving a *representation problem*.

You already know about mathematics as a way to model the world, though you may not have thought about it that way. An equation like  $F = ma$  is saying something about how the world works. It says that the amount of force ( $F$ ) in a collision (for example) is equal to the amount of mass ( $m$ ) of the moving object times its acceleration ( $a$ ). You might be able to imagine a world where that's not true—perhaps a cartoon world where a slow-moving punch packs a huge wallop. In that world, you'd want to use a different equation for force  $F$ .

The powerful thing about software representations is that they are executable—they have *behavior*. They can move, speak, and take action within the simulation that we can interpret as complex behavior, such as traversing a scene and accessing resources. A computer model, then, has a *structure* to it (the pieces of the model and how they relate) and a *behavior* to it (the actions of these pieces and how they interact).

Are there better and worse *physical structures*? Sure, but it depends on what you're going to use them for. A skyscraper and a duplex home each organize space differently. You probably don't want a skyscraper for a nuclear family with 2.5 children, and you're not going to fit the headquarters of a large multinational corporation into a duplex. Consider how different the physical space of a tree is from a snail—each has its own strength for the context in which it is embedded.

Are there better and worse information structures, *data structures*? Imagine that you have a representation that lists all the people in your department, some 50–100 of them sorted by last names. Now imagine that you have a list of all the people in your work or academic department, but grouped by role, e.g., teachers vs. writers vs. administrative staff vs. artists vs. management, or whatever the roles are in your department. Which representation is *better*? It depends on what you're going to do with it.

- If you need to look up the phone number of someone whose name you know, the first representation is probably better.
- If the artistic staff gets a new person, the second representation makes it easier to write the new person's name in at the right place.

### **Computer Science Idea: Better or worse structures depend on use**

A structure is better or worse than another structure depending on how it's going to be used – both for access (looking things up) and for change. How will the structure be changed in the future? The best structures allow you to find things quickly and also allow you to add or remove items quickly and easily.

\* \* \*

Structuring our data is *not* something new that appeared when we started using computers. There are lots of examples of structuring data and using representations in your daily life.

- Consider the stock listing tables that appear in your paper. For each stock (arranged vertically into rows), there is information such as the closing price and the difference from the day before (in columns). A *table* appears in the computer as a *matrix* which is also called a *2 dimensional array*.
- Our daughter, Katie, likes to create treasure hunts for the family, where she hides notes in various rooms (Figure 1.3). Each note references the next note in the list. This is an example of a *linked list*. Each note is a link in a chain, where the note tells you (links to) the next link in the chain. Think about some of the advantages of this structure: the pieces define a single structure, even though each piece is physically separate from the others; and changing the order of the notes or inserting a new note only requires changing the neighbors (the ones before or after the notes affected).

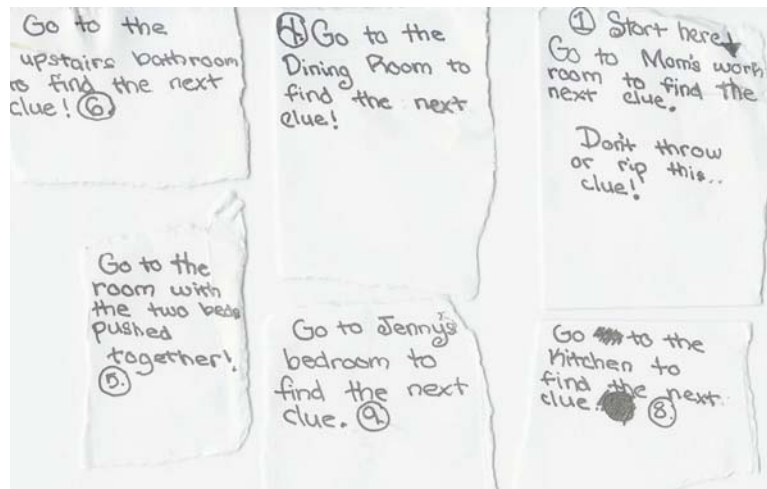


Figure 1.3: Katie's list of treasure hunt clues

- An organization chart (Figure 1.4) describes the relationships between roles in an organization. It's just a representation—there aren't really lines extending from the feet of the CEO into the heads of the Presidents of a company. This particular representation is quite common—it's called a *tree*. It's a common structure for representing *hierarchy*.

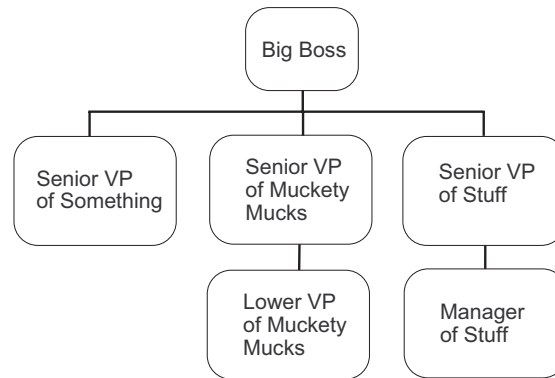


Figure 1.4: An organization chart

- A map (Figure 1.5) is another common representation that we use. The real town actually doesn't look like that map. The real streets have other buildings and things on them—they're wonderfully rich and complex. When you're trying to get around in the town, you don't want a satellite picture of the town. That's too much detail. What you really want is an *abstraction* of the real town, one that just shows you what you need to know to get from one place to another. We think about Interstate I-75 passing through Atlanta, Chattanooga, Knoxville, Cincinnati, Toledo, and Detroit, and Interstate I-94 goes from Detroit through Chicago. We can think about a map as *edges* or *connections* (streets) between points (or *nodes*) that might be cities, intersections, buildings, or places of interest. This kind of a structure is called a *graph*.

These data structures have particular *properties* that make them good for some purposes and bad for others. A table or matrix is really easy for looking things up (especially if it's ordered in some way). But if you have to insert something into the middle of the table, everything else has to move down to make room. When we're talking about space in the computer (*memory*), we're literally talking about moving each element in memory separately. On the other hand, inserting a new element into a linked list or into a graph is easy—just add edges (links) in the right places.

Why does the structure matter? It matters because of the way that computer memory works. If you think of memory as being a whole bunch of mailboxes in a row, each with its own address. Each mailbox stores exactly one thing. In reality, that one thing is a binary pattern, but we can interpret it any way we want, depending on the encoding. Maybe it's a number or maybe it's a character.

A table (a matrix or an array) is stored in consecutive mailboxes. So, if

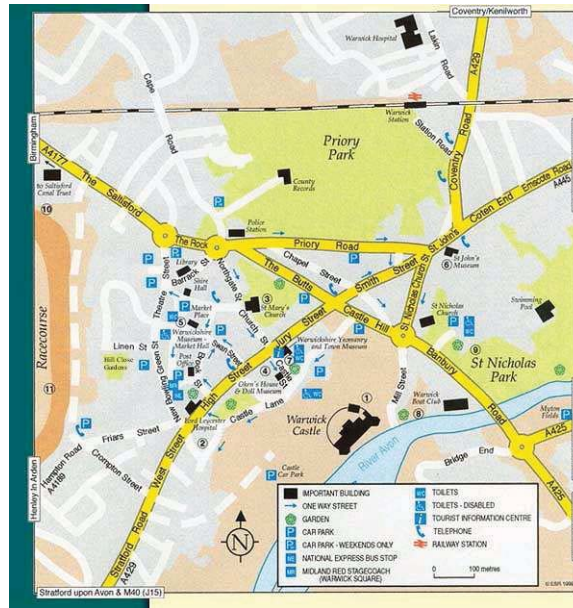


Figure 1.5: A map of a town

you have to put something into the middle of a table, you have to move the things already in there somewhere else. If you put something new where something old used to be, you end up over-writing the something old.

To make it clear, let's imagine that we have a table that looks something like this:

Name	Age	Weight
Arnold	12	220
Kermit	47	3
Ms. Piggy	42	54

Let's say that we want to add "Fozzie" to the list, who's 38 and weighs 125 pounds. If we add him alphabetically he would go below Arnold and above Kermit, but if just put him after Arnold, we would over-write Kermit. So, the first thing we have to do is to make room for Fozzie at the *bottom* of the table. (We can simply add more mailboxes after the table.)

Name	Age	Weight
Arnold	12	220
Kermit	47	3
Ms. Piggy	42	54

Now we have to copy everything down into the new space, opening up a spot for Fozzie. We move Ms. Piggy and her values into the bottom space, then Kermit into the space where Ms. Piggy was. That's two *sets* of data that we have to change, with three values in each set.

Notice that that leaves us with Kermit's data duplicated.

Name	Age	Weight
Arnold	12	220
Kermit	47	3
Kermit	47	3
Ms. Piggy	42	54

Once we add the information for Fozzie in the space where Kermit was originally the table will look like this:

Name	Age	Weight
Arnold	12	220
Fozzie	38	125
Kermit	47	3
Ms. Piggy	42	54

Now let's compare that to a different structure, one that's like the treasure trail of notes that Katie created. We call that a linked list representation. Consider a note (found in a bedroom) like the following:

"The next note is in the room where we prepare food."

Let's think about that as a note *in* the bedroom that *references* (says to *go to*) the kitchen (Figure 1.6).



Figure 1.6: Bedroom note refers to the kitchen

In terms of memory mailboxes, think about each note as having two parts: a current location, and where the *next* one is. Each note would be represented as two memory mailboxes (Figure 1.7).



Figure 1.7: Memory mailboxes for bedroom note

So let's imagine that Katie has set up a trail that looks like the one in Figure 1.8.

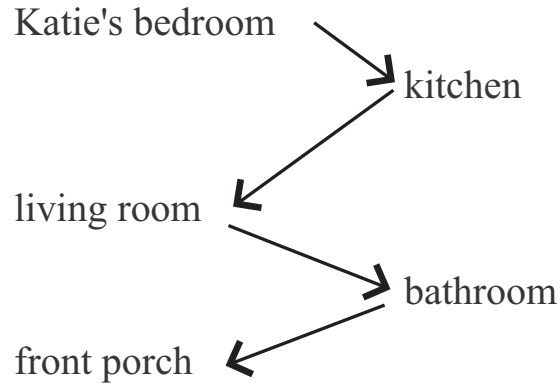


Figure 1.8: Treasure hunt trail

Now, she changes her mind. Katie's bedroom shouldn't refer to the kitchen; her bedroom should point to Matthew's bedroom as the next location. How do we change that? We simply put a new note in Katie's bedroom that refers to Matthew's bedroom as the next location. We move the note that was in Katie's bedroom to Matthew's bedroom. It will still refer to the kitchen as its next location. None of the other notes need to move or change as shown in Figure 1.9.

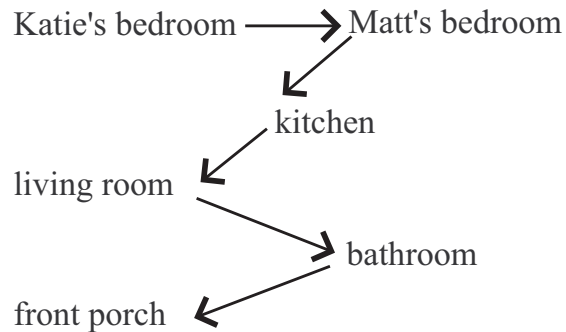


Figure 1.9: Modified treasure hunt trail

In terms of memory mailboxes we added a new memory mailbox with a current location of Matt's bedroom and a next location equal to the original next location of the first memory mailbox in the trail, the kitchen. Then

we changed the next location of the first memory mailbox in the trail to refer to Matthew’s bedroom.

Adding to a linked list representation is much easier than adding to a table, especially when you’re adding to the middle of the table. But there are advantages to tables, too. They can be faster for looking up particular pieces of information.

Much of this book is about the trade-offs between different data structures. Each data structure has strengths that solve some sets of problems, but the same data structure probably has weaknesses in other areas. Each choice of data structure is a trade-off between these strengths and weaknesses, and the choices can only be made in the context of a particular problem.

These data structures have a *lot* to do with our wildebeests and villagers.

- The visual structure of villagers and wildebeests (e.g., how legs and arms attach to bodies) is typically described as a tree or graph.
- Tracking which villager is next to do something (e.g., move around) is a queue.
- Tracking all of the wildebeests in the stampede is often done in a *list* (like a linked list).
- The images to be used in making the villagers wave or the wildebeests run are usually stored in a list.

## 1.2 Why Java?

Why is this class taught in Java, instead of another language like Python?

- Overall, Java is faster than Python (and definitely faster than Jython). We can do more complex things faster in Java than in Python.
- Java is a good language for exploring and learning about data structures. It makes it explicit how you’re connecting data through *references*.
- More computer science classes are taught in Java than Python. So if you go on beyond this class in data structures, knowing Java is important.
- Java has “resume-value.” It’s a well-known language, so it’s worth it to be able to say, even to people who don’t really know computer science, that you know Java. This is important—you’ll learn the content better if you have good reason for learning it.

- Java has a huge library of pre-built classes that make creating programs easier. For example, it has components for creating professional looking graphical user interfaces, for reading information from files or from networks, and for working with relational databases.

## Getting Java Set-Up

You can start out with Java by simply downloading a Standard Edition (SE) *JDK (Java Development Kit)* from <http://www.java.sun.com> for your computer. With that, you have enough to get started programming Java. However, that's not the easiest way to *learn* Java. In this book, we use *DrJava* which is a useful *IDE (Integrated Development Environment)*—a program that combines facilities for editing, compiling, debugging, and running programs. DrJava is excellent for learning Java because it provides an INTERACTIONS PANE where you can simply type in Java code and try it out.

If you'd like to use DrJava, follow these steps:

- Download DrJava from <http://www.drjava.org> and install it.
- Download *JMusic* from <http://jmusic.ci.qut.edu.au/> and install it
- Make sure that you grab the `media-source` and `java-source` from the book website.
- You'll need to tell DrJava where to find the classes that we created for this book and the libraries for JMusic. You use the Preferences in DrJava (see Figure 1.10) to add the `java-source` directory, the *JMusic jar file* (`jmusic.jar`), the `instruments` directory (`inst`), and all the jar files (`jmf.jar`, `javazoom.jar`, and `sound.jar`) in the `java-source` directory, (Figure 1.11) to the list of locations to look for unknown classes. This list of places to look for unknown classes is known as the *classpath*. Once you have added all items to the classpath click on APPLY and OK.

### **Making It Work Tip: Keep all your Java files in your `java-source` directory**

Once you add `java-source` to your classpath this is the location DrJava will look for classes that you use that aren't part of the Java language. If you move your source code to a different directory it won't be found. (Figure 1.11).

\* \* \*

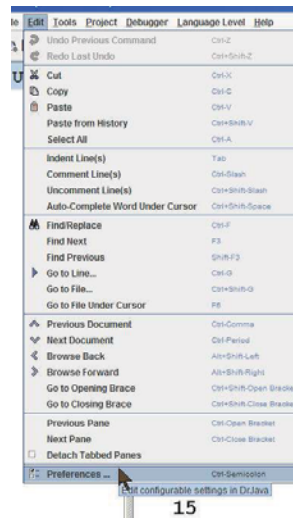


Figure 1.10: Open the DrJava Preferences by clicking on Edit and then Preferences

Once you start DrJava, you'll have a screen that looks like Figure 1.12.

If you choose *not* to use DrJava, that's fine. Set up your IDE and be sure to install JMusic and set up your classpath to access the JMusic jar (`jmusic.jar`), the `inst` directory, the `java-source` directory, and the jars (`jmf.jar`, `javazoom.jar`, and `sound.jar`) in the `java-source` directory. This book will assume that you're using DrJava and will describe using classes from the Interactions Pane, but you can easily create a class with a main method instead.

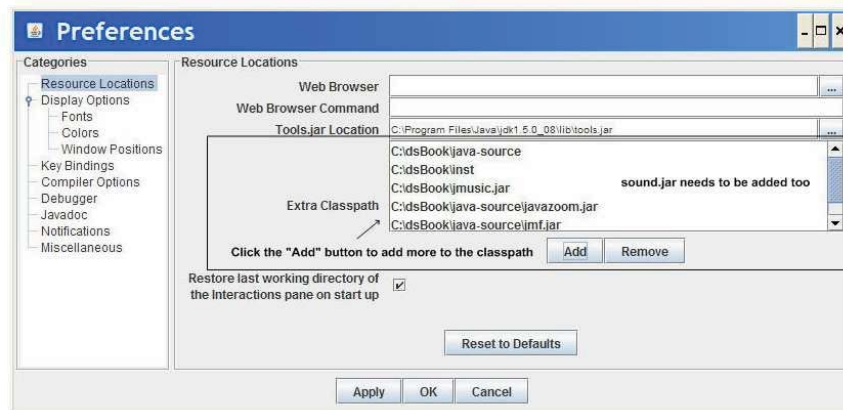


Figure 1.11: Adding `java-source`, the `jMusic` libraries, and the jars in `java-source` to the classpath in DrJava

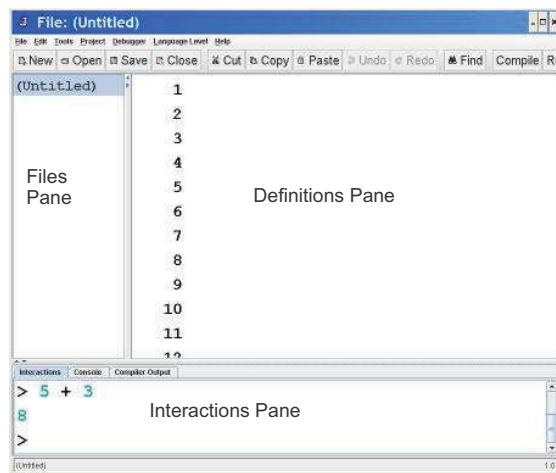


Figure 1.12: Parts of the DrJava window

## 2 Introduction to Java

### Chapter Learning Objectives

- Explain why Java is relevant to modeling and simulation.
- Give the basic syntax details for variables, arrays, iteration, and conditionals.
- Explain how to create a class and a subclass.
- Give the user a brief introduction to the manipulation of pictures, sounds, and music.

### 2.1 What's Java about?

Java programs contain interacting *objects*. In *object-oriented programming*, the programmer cares about more than just *specifying* a process. In other languages, like Python or Visual Basic, you mostly tell the computer “First do this, then do that.” In object-oriented programming, you care about who (or what) does each task in the process, and how the overall process *emerges* from the interaction of different objects. The software engineering term for this is *responsibility-driven design*—we don’t just care about how the process happens, we care about who (which object) does which part of the process.

An example might help. When you go to the movies there are people that sell tickets, people that take the tickets, people that sell drinks, popcorn, and candy, and people that clean the theaters. Each job has a set of responsibilities. Each job has skills that people doing the job must have and information that people doing the job need to know. People that sell tickets need to be able to process payment and need to know which movies are sold-out. People that take the tickets need to know where the theaters are in order to direct people to the theater. A job *classification* identifies the required skills for the job.

In Java we create *classes* that identify the things that objects of the class can do (called *methods*) and the information (called *fields*) that objects of the class need to keep track of. When creating a Java program to model a movie theater we would create a class for each job classification. We would also create classes for other objects at a movie theater like

movie, food, ticket, theater, etc. Each movie has a name and a length. A movie can be shown several times a day so we will need an object for each showing. Each showing will have a date and start time. There could be private showings as well as public ones. A private showing will have contact information as well as the usual showing information. In object-oriented programming we call the relationship between movie and a showing of a movie an *association* or a *has-a* link which is shown as a solid line on figure 2.1. The relationship between showing and private showing is an *inheritance* or a *is-a-type-of* relationship which is shown as an open triangle pointing toward the *parent class* in figure 2.1. In inheritance the *child class* (private showing) inherits data and behavior from the *parent class* (movie showing). Figure 2.1 is a *UML class diagram* which is used to show classes and the relationships between them. Each class is shown in a box with the name of the class at the top of the box. Under the class name you can show the fields (the data each object will have).

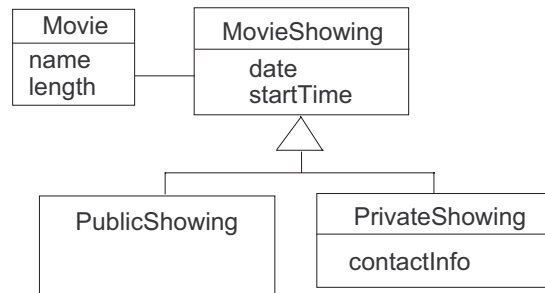


Figure 2.1: UML class diagram of movie types and movie showings

Object-oriented programming dates back to a programming language called *Simula*, which was a programming language for creating simulations. The idea was to describe the world that you cared about in the Simula language, e.g. how customers work their way through a store, how material flows through a factory, how deer and wolves balance each other ecologically in an ecosystem. That description is called a *model*. When Alan Kay used Simula in the late 1960's, he realized that all programs can be thought of as modelling some world (real or imaginary) and that all programming is about simulation. It was this insight that led to his programming language *Smalltalk* and our current understanding of object-oriented programming. Java is an object-oriented programming language and is a descendant of Smalltalk.

Thinking about programming as modelling and simulation means that you *should* do responsibility-driven design—you need to spread control over what happens in the overall process across many objects. That's the way that the real world works. Setting aside theological arguments, there

is no great big **for** loop telling everything in the real world to take another time step. You don't write one big master program in Java—your program arises out of the interaction of lots of objects, just like the real world. Most importantly, in the real world, no *one* object **knows** everything and can **do** everything. Instead, in the real world and in Java, each object has things that it *knows* and things that it can *do*.

## 2.2 Basic (Syntax) Rules of Java

This section contains the basic rules for doing things in Java. We'll not say much about classes, fields, and methods here—we'll introduce the syntax (rules) for those as we need them. These are the things that you've probably already seen in other languages.

### Declarations and Types

If your past programming experience was in a language like Python, Visual Basic, or Scheme, the trickiest part of learning Java will probably be its *types*. All variables and values in Java have a type. You must declare a variable before you can use it. To declare a variable specify the type and the name of the variable.

```
TYPE NAME;
```

You can also initialize the value of the variable to the result of an expression when you declare it.

```
TYPE NAME = EXPRESSION;
```

The type of a variable can be either the name of a class or one of the primitive types. Java stores integer numbers, floating point numbers, single characters, and boolean values as primitives (not objects). It does this in order to make calculations quicker.

When primitive variables are declared, memory is set aside to hold a value of that type and the variable name is associated with that memory. When you use the name again in your program the name will be used to find the memory that stores the value and the value will be substituted for the name in expressions.

In the following Java code we are declaring a variable named `a` that can hold integer values. We then set the value of `a` to 5 and then add 7 to the value in `a`. The value stored in `a` was 5 so the result is 12. If you enter a statement in the interactions pane of DrJava and do not add a semicolon at the end of the statement DrJava will print out the result of the statement.

```
> int a = 5;
> a + 7
12
```

You can also print the output of the result of an expression using the following Java code `System.out.println(EXPRESSION)` which will output the result of the expression followed by the end of a line.

```
> System.out.println(a + 7);
12
```

You can only declare a variable once in the interactions pane since declaring it assigns memory to it and associates the name with that memory. You can't reserve two different sections of memory with the same name. Which one would you mean when you used the name? But, you can change the value of the variable (the value in memory) as often as you want.

```
> int a = 6;
Error: Redefinition of 'a'
> a = 6;
> System.out.println(a);
6
> a = -8;
> System.out.println(a);
-8
```

How much memory is used for a variable? Some types take up more memory than others. The type **int** is used to store positive and negative integers such as 352 and -20. The amount of space used to store **int** variables is 32 bits long and the values are stored using two's complement notation. Bit is short for binary digit and each bit can store the value 0 or 1. The type **char** is used to store single characters such as 'a' or 'T'. A **char** variable is 16 bits long and the value is stored using the Unicode format. The length of the type **boolean** is not specified, but it could be stored in just one bit with 0 for **false** and 1 for **true**. The values **true** and **false** are literals in Java.

```
char firstLetter = 'a';
System.out.println(firstLetter);
'a'
> boolean flag = true;
> System.out.println(flag)
true
```

In Java there are two types that can be used to represent floating point (decimal) numbers such as 25.321 or -3.04. The types are **float** and **double**. The type **double** can be used to store more significant digits than the type **float** since a **double** is stored in 64 bits and a **float** is stored in 32 bits. A floating point variable is stored in IEEE 754 floating point format. As you can see below you must use the letter 'f' after a number with a decimal in it if you want it to be of the type **float** because otherwise the compiler will assume that the number is a **double**.

```
> float f = 13.2f
> f
13.2
```

Notice that you don't get what you might expect from the following addition (26.431). Floating point numbers are not exactly represented in computers and calculations involving floating point numbers do not give exact results.

```
> double d;
> d = 13.231;
> d
13.231
> d + f
26.43099980926514
```

You can't put a floating point number into an integer because an integer doesn't store a fractional part.

```
> a = f
Error: Bad types in assignment
```

You can tell the computer to make a double value fit into an integer variable by throwing away the fractional part. Use a *cast* to do this. A cast is indicated by specifying the type to cast to inside of a pair of parentheses in front of the value or expression to be cast. This is like casting clay to form a new shape.

```
> a = (int) f
13
```

## Strings

You can represent a series of characters using the class `String`. Note that the `String` object is not just an array of characters, like it is in some languages. You can assign a value to a `String` object by enclosing a series of characters in a pair of double quotes. The class `String` is defined as part of the Java language.

```
> String s = "This is a test";
> System.out.println(s);
"This is a test"
```

There are many things you can do to `String` objects in Java. You can create a new `String` that has the same characters as the original `String`, but in the new `String` object all the characters are lowercase. In Java `String` objects are *immutable*. Immutable means that they can't change. If you try changing a `String` in Java you get back a new `String` object.

```
> String start = "UPPERCASE LETTERS ARE LIKE SHOUTING";
> String lower = start.toLowerCase();
> System.out.println(lower):
uppercase letters are like shouting
> System.out.println(start)
UPPERCASE LETTERS ARE LIKE SHOUTING
```

You can look for something in the `String`. If the thing isn't found in the `String`, it returns the value `-1`. If the thing is found in the `String`, it returns the position that it was found at. The first character in a string is at position `0`.

```
> int position = start.indexOf("IS")
> System.out.println(position);
-1
> int posU = start.indexOf("U");
> posU
0
```

There are many more things that you can do with `String` objects. We recommend that you take a look at the documentation for the `String` class if you want to learn more. The `String` class is in the `java.lang` package. Java contains hundreds of classes. Classes are organized into *packages*. A package is a collection of related classes. The classes in the `java.lang` package are the core classes in Java. The classes in package `java.io` handle input and output. The classes in the `java.net` package work with networks. Go to [java.sun.com](http://java.sun.com) and find the *API* (application programmer interface) for the version of Java you are using. Then click on the package name `java.lang` to see all the classes in that package. Click on the `String` class (Figure 2.2) and scroll down to the methods section of the documentation, this is alphabetical list of the things you can do with strings.

## Object Variables and Primitive Variables

When you declare a type using a class name in Java then a variable is created in memory that has as its value a way to find the actual object of that class in memory. This is called a *reference* to an object. This is different from how variables of primitive types work. When you declare a primitive type the memory associated with the variable name holds the value. So if we declare an `int` variable `a` to have a value of `13` and a `String` variable `s` to have a value of `"This is a test"` we would get the memory map shown in Figure 2.3.

Why don't we just store the object in the memory associated with the variable like we do for primitive types? The problem is that when you declare a variable with a type that is a class name then that variable can be used to refer to an object of that class name or an object of any subclass (child) of that class. An object of a subclass can be larger than an object of



Figure 2.2: Showing the API for the String class

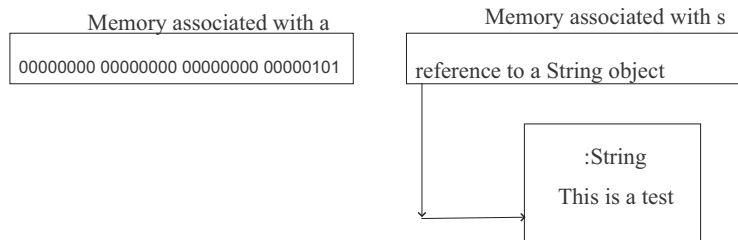


Figure 2.3: Showing memory for an int variable and a String variable

the class. For example, the String class in Java is a child of the Object class. You can declare a variable `o` that is of the type Object and set the value to a new String object.

```
> Object o = new String("hi");
> System.out.println(o);
"hi"
```

Since a object of the String class is bigger than an object of the Object class it wouldn't fit in the variable if we had only allocated enough memory for an Object. So instead when we declare a variable with a type that is a class name memory is allocated that stores a reference to an object. The reference is used to find the actual object in memory.

You can think of a reference to an object as being like a tracking number on a package. A package tracking number is the same size regardless of the size of the package and is used to find the actual package. The reference

is used to find the actual object in memory. The class allocates memory for an object when you create it since the class knows the information (fields) defined for each object of the class.

There is a special value in Java called **null** that indicates that a reference doesn't refer to any object. So, if I declare a new String variable but do not assign it to a valid String object I get the value **null** when I print the variable.

```
> String s1;  
> System.out.println(s1);  
null
```

## Assignment

VARIABLE = EXPRESSION

The equals sign (=) is used for assignment. The VARIABLE should be replaced with a declared variable, or (if this is the first time you're using the variable) you can declare it in the same assignment, e.g., **int** a = 12;. If you want to create an object (not a *literal* like the numbers and strings in the last section, you use the term **new** with the name of the class (and perhaps include input for use in initializing the object fields). For example, we have created a Picture class that you can use to display (show) and modify digital pictures.

```
> Picture p = new Picture(FileChooser.pickAFile());  
> p.show();
```

## Semicolons

Java statements are ended by a semicolon. Statements can take up several lines. You can think of a Java statement like an English sentence, except that it ends in a semicolon, not a period. So you do need a semicolon at the end of an assignment statement since you are saying set the value of the variable to the result of the expression, which is a complete sentence.

### Common Bug: Forgetting a semicolon at the end of a statement

If you forgot to add a semicolon at the end of a statement the compiler will report: "Error: ';' expected". This is a very easy error to fix. Just click on the highlighted line in the compiler output and it will take you to the statement that is missing the semicolon. Add the semicolon and recompile. Occasionally the actual error is above where it shows it in the code so check above if the current statement looks okay.

\* \* \*

## Conditionals

You can conditionally execute a statement when an expression is true.

```
if (EXPRESSION)
    STATEMENT

> int x = 3;
> if (x == 3)
    System.out.println("x is 3");
x is 3
```

Notice that while `=` is used to assign a value to a variable, `==` is used to test for equality.

You can also execute a block of statements when an expression is true. A block of statements is enclosed in a pair of curly braces.

```
if (EXPRESSION) {
    STATEMENT
    STATEMENT
    ...
}
```

In the example above you may notice that the open curly brace is at the end of the line after the `if`. This is the Java style standard, but some people prefer the open curly brace at the beginning of the next line. It doesn't matter to Java if you put the open curly brace at the end of a statement or on the next line. Some people find it easier to match curly braces when the open curly brace is at the beginning of an empty line. But, this does make the code longer. We will use the Java style standard in this book.

```
int y = 2;
if (y < 2)
    System.out.println("y is >= 2");
    System.out.println("done");
System.out.println("after if");
```

Try the example above. Do you get the output you expect? If you have more than one statement following an `if` that you want executed when the condition is true, be sure to put the statements in a block defined by a pair of open and close curly braces. The *indention* (extra spaces at the beginning of a statement) is ignored by Java. We just indent code to make it easier for us to see the blocks.

**Debugging Tip: Always use a block of statements with conditionals**  
Even though you don't need to put a single statement following an `if` inside

a pair of curly braces, it will make your code safer if you do. A common bug is to add more than one statement after an `if`, but forget to add the curly braces to form a block of statements. When the code executes, only the first statement following the `if` is executed as part of the conditional.

An expression in Java is pretty similar to a logical expression in any other language. One difference is that a logical *and* is written as `&&`, and an *or* is written as `||`.

You can execute one statement or block of statements when the expression is true, and a different statement or block of statements when the expression was false using an `if` and `else`.

```
if (EXPRESSION)
    THEN-STATEMENT OR BLOCK OF STATEMENTS
else
    ELSE-STATEMENT OR BLOCK OF STATEMENTS

int z = 3;
if (z < 3) {
    System.out.println("z < 3");
}
else {
    System.out.println("z >= 3");
}
```

### Common Bug: Watch out for extra semicolons

Remember that a semicolon is one way to end a Java sentence. A block is another way. So we don't need to add a semicolon at the end of a block, but Java won't care if you do. But, if you add a semicolon in the middle of a conditional statement (after the boolean expression) you end the statement which can lead to some very unexpected results.

## Arrays

To declare an array, you specify the *type* of the elements in the array followed by open and close *square brackets*. (In Java all elements of an array have the same type.) `Picture []` declares an array of type `Picture`. So `Picture [] myArray;` declares `myArray` to be a variable that refers to an array of `Pictures`.

To actually create the array, we might say something like `new Picture[5]`. This declares an array of references to five pictures. This does *not* create the pictures, though! Each of those have to be created separately. Java indices start with zero, so if an array has five elements, the minimum index

is zero and the maximum index is four. The fifth array reference (as seen below) will result in an error—`ArrayIndexOutOfBoundsException`.

```
> Picture [] myArray = new Picture[5];
> Picture background = new Picture(800,800);
> FileChooser.pickMediaPath();
> myArray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
> myArray[1]=new Picture(FileChooser.getMediaPath("jungle.jpg"));
> myArray[2]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
> myArray[3]=new Picture(FileChooser.getMediaPath("flower1.jpg"));
> myArray[4]=new Picture(FileChooser.getMediaPath("flower2.jpg"));
> myArray[5]=new Picture(FileChooser.getMediaPath("butterfly.jpg"));
ArrayIndexOutOfBoundsException:
  at java.lang.reflect.Array.get(Native Method)
```

You can also initialize the contents of an array when you declare it. The size of the array will be the number of items inside the open and close curly braces. Separate the array items with commas. Arrays are objects and they know their length. You can get the number of items in an array using `NAME.length`. Note that there are not any parentheses after length, since it is a *public field* and not a method. Public means that anyone (any class) can see the field and directly access the value using `objectRef.fieldName`.

```
> numArray.length
4
```

## Iteration

As of Java 1.5 (also called Java 5) you can use a for-each loop to loop through the elements of an array or collection and repeat a statement or block of statements.

```
for (TYPE NAME : ARRAY)
    STATEMENT OR BLOCK OF STATEMENTS
```

A concrete example will help to explain this. I can create an array of integers and then total them.

```
> int[] gradeArray = {90, 80, 85, 70, 99};
> total = 0;
> for (int grade : gradeArray)
    total = total + grade;
> System.out.println(total);
424
```

This will loop through each element of the array. Each time through the loop the grade variable will be set to the next array element. The current value of the grade variable will be added to the total. So, the first time the grade variable will have the value 90, and the second time it will have

the value 80, and so on until the last time through the loop it will have the value of 99.

You can also use a while loop to repeat a statement or block of statements as long as an expression is true.

```
while (EXPRESSION)
    STATEMENT OR BLOCK OF STATEMENTS
```

There is a **break** statement for exiting the current block. There is also a **continue** statement that will jump to the next execution of the loop.

Probably the most confusing iteration structure in Java is the **for** loop. It really combines a specialized form of a **while** loop into a single statement.

```
for (INITIAL-EXPRESSION;
     CONTINUING-CONDITION;
     ITERATION-EXPRESSION)
    STATEMENT OR BLOCK OF STATEMENTS
```

A concrete example will help to make this structure make sense.

```
> for (int num = 1 ; num <= 10 ; num = num + 1)
    System.out.print(num + " ");
1 2 3 4 5 6 7 8 9 10
```

The first thing that gets executed *before anything inside the loop* is the INITIAL-EXPRESSION. In our example, we're creating an integer variable `num` and setting it equal to 1. We'll then execute the loop, testing the CONTINUING-CONDITION before each time through the loop. In our example, we keep going as long as the variable `num` is less than or equal to 10. Finally, there's something that happens *after* each time through the loop – the ITERATION-EXPRESSION. In this example, we add one to `num`. The result is that we print out (using `System.out.print`, which is the same as `print` in many languages) the numbers 1 through 10. The expressions in the **for** loop can actually be several statements, separated by commas. Notice that `System.out.println` printed the result of the expression and then moved to a new line, while `System.out.print` prints the value of the expression and stays on the same line.

The phrase `VARIABLE = VARIABLE + 1` is so common that a short form has been created (`VARIABLE++`).

```
> for (int num = 1 ; num <= 10 ; num++)
    System.out.print(num + " ");
```

## Strings versus Arrays and Substrings

A Java *string* is not an array of characters like it is in some languages. But, you can create a string from an array of characters. Characters are

defined with single quotes (e.g., 'a') as opposed to double quotes (e.g., "a") which define strings.

```
> char characters[]={'B','a','r','b'}
> characters
[C@1ca209e
> String wife = new String(characters)
> wife
"Barb"
```

You cannot index a string with square brackets like an array. You can use the substring method on a string to retrieve part of the string (*substrings*) within the string. The substring method takes a starting and ending position in the string, starting with zero. It does not include the character at the ending position.

```
> String name = "Mark Guzdial";
> name
"Mark Guzdial"
> name[0]
Error: 'java.lang.String' is not an array
> name.substring(0,0)
""
> name.substring(0,1)
"M"
> name.substring(1,1)
""
> name.substring(1,2)
"a"
```

## 2.3 Using Java to Model the World

We have talked about the value of objects in modeling the world, and about the value of Java. In this section, we use Java to model some objects from our world. Let's consider the world of a student.

### A Problem and Its Solution: What should I model?

When you model something in the world (real or virtual) as an object, you are asking yourself "What's important about the thing that I am trying to model?" Typically, that question is answered by considering what you want from the model. Why are you creating this model? What's important about it? What questions are you trying to answer?

If you are creating a model of students, then the question that you want to answer determines what you model and how you model it. If you want

to create a model in order to create a course registration system, then you care about the students as people with class sections. If you wanted to model students to answer questions about their health (e.g., how dormitory food impacts their liver, or how lack of sleep impacts their brains), then you want to model students as biological organisms with organs like livers.

For our purposes in this chapter, let's imagine that we are modeling students in order to explore registration behavior. Given the previous, that means that we care about students as people. At the very least then, we want to define a class `Student` and a class `Person`.

### A Problem and Its Solution: How much should I model?

Always try for the minimal model. Model as little as possible to answer your question. Models grow in complexity rapidly. The more attributes that you model, the more that you have to worry about later. When you model, you are always asking yourself “Did I deal with all the relationships between all the variables in this model?” The more variables you model, the more relationships there are to consider. Model as little as you need to answer your question.

We are going to define the class `Student` as a *subclass* of `Person`. There are several implications of that statement.

- Class `Person` is a *superclass* of `Student`. We also say that `Person` is the *parent class* of `Student`, and `Student` is the *child class* of `Person`. We should be able to think about the subclass, `Student`, as being “a kind of” (sometimes shortened to *kind-of*) the superclass. A student is a kind of person—that's true, so it's a reasonable superclass-subclass relationship to set up.
- All *fields* or *instance variables* of the class `Person` are automatically in instances of the class `Student`. That does not mean that all those variables are directly accessible—some of them may be *private* which means that they are defined in the parent class, they are directly accessible in the parent class, but they are not directly accessible in the methods of the subclass.
- All *methods*, except private methods, in the superclass, `Person` are automatically callable in the subclass `Student`.

Here's an initial definition of the class `Person`.

*Program*  
*Example #1*

Example Java Code: **Person class, starting place**

```
public class Person {
    public String name;
}
```

We have created a **public** class named `Person` that has a public field called `name` of type `String`. The keyword **public** is used to specify what classes have access to this class and this field. This is called specifying the *visibility*. Using **public** visibility means that all other classes have access to this class and field.

That's enough to create an instance (object) of class `Person` and use it. Here's how it will look in DrJava:

```
> Person fred = new Person();
> fred.name
null
> fred.name = "Fred";
> fred.name
"Fred"
```

Public fields can be directly accessed and modified from code in any class using *dot notation*. Just use the variable name followed by a period and then the name of the field. There's an implication of this implementation of `Person`—we can change the name of the person in the interactions pane. Should we be able to do this?

```
> fred.name = "Mabel";
> fred.name
"Mabel"
```

If we want to control direct name changes so that they can only happen within the `Person` class, we should make the field `name` *private*. *Private visibility* means that only code in the same class has direct access. So, **private** fields can't be directly modified by code in other classes. So, to get to the name in code that is outside the `Person` class, we can use a public *accessor* method (sometimes called a *getter* method because it lets us get the value of a private field. To change the name from outside the class we would use a public *mutator* or *modifier* method (sometimes called a *setter* method because it lets us set the value of a private field. These methods could check whether it's appropriate to get and/or set the variable value and make sure the value is acceptable.

Example Java Code: **Person, with a private name and public accessor and modifier methods**

*Program  
Example #2*

```
public class Person {
```

```

    private String name;

    public void setName(String someName) {
        this.name = someName;
    }

    public String getName() {
        return this.name;
    }
}

```

**How it works:** The field `name` is now **private** meaning that it can be directly manipulated only inside the class `Person`, (using `this.name`). The mutator method `setName` takes a `String` as input, then sets the name to that input. Since `setName` doesn't return any value, its return value is **void**—literally, nothing. The accessor method `getName` does return the value of the variable `name` so it has a return value of `String`. Notice that both of these methods refer to `this.name`. The keyword **this** is a special variable meaning the object that has been told to `getName` or `setName`. The phrase `this.name` means “The variable name that is in the object **this**, the one that was told to execute this method.”

Now, we manipulate the field name using these methods—because we can't access the variable directly anymore. Within `getName` and `setName` in this example, the variable **this** means `fred` — they are two variable names referencing the exact same object.

```

> Person fred = new Person();
> fred.setName("Fred");
> fred.getName()
"Fred"

```

This works well for getting and setting the name. Let's consider what happens when we first create a new `Person` instance.

```

> Person barney = new Person();
> barney.getName()
null

```

Should “barney” have a **null** name? The value **null** means that this variable does not yet reference a `String` object. How do we define an instance of `Person` such that it automatically has a value for its name? That would be the way that the real world works, that baby people get names at birth.

We can make that happen by defining a *constructor*. A constructor gets called when a new object (instance) is created. A constructor has the same name as the class itself. It can take input (parameters), so that we

can create an object with certain values. Constructors don't *have* to take input—it's okay to just create an instance and have pre-defined values for variables.

Example Java Code: **Person, with constructors**

*Program  
Example #3*

```
public class Person {
    private String name;

    public Person() {
        this.setName("Not-yet-named");
    }

    public Person(String name) {
        this.setName(name);
    }

    public void setName(String someName) {
        this.name = someName;
    }

    public String getName() {
        return this.name;
    }
}
```

**How it works:** This version of class `Person` has two different constructors. One of them takes no inputs, and gives the name field a predefined, default value. The other takes one input—a new name to be given to the new object. It's okay to have multiple constructors as long as they can be distinguished by the inputs (this is called *overloading*). Since one of these takes nothing as input, and the other takes a `String`, it's pretty clear which one we're calling when we create `Person` objects. The `'''` in the code below is the beginning of a *comment*. A comment is ignored by the computer but there to help explain your code to other people.

```
> Person barney = new Person("Barney")
// Here, we call the constructor that takes a string.
> barney.getName()
"Barney"
> Person wilma = new Person()
// Here, we call the constructor that doesn't take an input
> wilma.getName()
"Not-yet-named"
```

```
> wilma.setName("Wilma")
> wilma.getName()
"Wilma"
```

If we specify parameters that don't match the current constructors we will get an error.

```
> Person agent99 = new Person(99)
NoSuchMethodException: constructor Person(int)
```

This is telling us that there is no constructor in the Person class that takes an integer value.

### Discourse Rules for Java

In an American “Western” novel or movie, there are certain expectations. The hero carries a gun and rides a horse. The hero never uses a bazooka, and never flies on a unicorn. Yes, you can make a Western that has a hero with a bazooka or a unicorn, but it's considered a weird Western—you've broken some rules.

Those are *discourse rules*—the rules about how we interact in a certain genre or setting. They are not laws or rules that are enforced by some outside entity. They are about expectations.

There are discourse rules (conventions) in all programming languages. Here are some for Java:

- Class names start with a capital letter and the rest of the first word is lowercase (like String). Variables, fields, parameters, and method names all start with a lowercase word. If a name has more than one word in it the first letter of each new word is capitalized (like FileChooser. This makes it easier to read a variable name that has more than one word in it.
- Class names are never plural. If you want more than one instance of a class, you use an array or a list.
- Class names are typically nouns, not verbs.
- Methods should describe what objects know how to do.
- Accessors and modifiers are typically named “set-” and “get-” followed by the name of the field.

### Defining toString

What is an instance (object) of the classPerson? What do we get if we try to print its value?

```
> Person barney = new Person("Barney")
> barney
Person@63a721
```

That looks like Barney is a Person followed by some expletive or code. This code is the hashcode which is a unique identifier for an object. We can make the display of Barney look a little more reasonable by adding the `toString` method to our Person class. When we try to print an object (by simply displaying its value in the Interactions Pane in DrJava, or when printed from a program using `System.out.println`), the object is converted to a string and printed. The method `toString` does that conversion. The Person class inherits the `toString` method from the class `Object`. How do we know that, you might ask? Well, if you don't specify what class you are inheriting from when you declare a new class it automatically inherits from the `Object` class. All classes in Java inherit at some level (parent, grandparent, great-grandparent, etc) from the `Object` class. The `toString` method in `Object` prints the class name followed by the hashcode of the object. By providing a `toString` method in the Person class we *override* the parents method with the same name and parameter list. Overriding a parent's method means that the child's method will be called instead of the parent's method. This happens because the method to execute is determined at run-time based on the class that created the object that the method is being called on. This run-time finding of the method to execute based on the class that created the object is a form of *polymorphism*. Polymorphism means many forms and it allows the same function name to be used for different data types. If that class that created the object that you are calling the method on has the method, it will be called. If the class that created the object doesn't have the method the parent's method will be executed (if it has the method). If the parent class doesn't have the method it will keep looking up the inheritance chain to find the method. The method will be found since the code wouldn't have compiled if it didn't exist at some level in the inheritance tree.

#### Example Java Code: `toString` method for Person

*Program  
Example #4*

```
public String toString() {
    return "Person named " + this.name;
}
```

**How it works:** The method `toString` returns a `String`, so we declare the return type of `toString` to be `String`. We can stick the words "Person named" before the actual name using the `+` operator—this is called *string concatenation*.

Now, we can immediately print Barney after creating him, and the output is reasonable and useful.

```
> Person barney = new Person("Barney")
> System.out.println(barney);
Person named Barney
```

### Defining Student as subclass of Person

Now, let's define our class Student. A student is a kind of person—we established that earlier. In Java, we say that Student **extends** Person—that means that Student is a subclass of Person (Figure 2.4). It means that a Student is everything that a Person is. Notice in figure 2.4 that Person is a subclass of Object as we mentioned earlier.

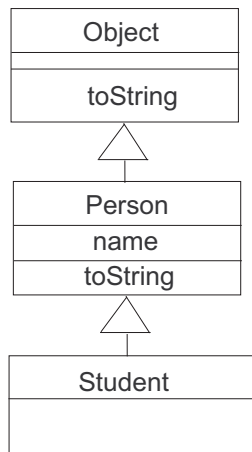


Figure 2.4: A UML class diagram for Person and Student

How should Student instances be different? Let's say that a Student has an identifier. That's not particularly insightful, but it is likely true.

*Program  
Example #5*

Example Java Code: **Student class, initial version**

```
public class Student extends Person {
    private int id;
    public int getId() { return this.id;}
    public void setId(int theId) {this.id = theId;}
```

```
}

```

**How it works:** Assuming that the identifier, `id`, will fit within the bounds of an integer `int`, this isn't a bad way to go. If the identifier might be too large, or if we would want to record dashes or spaces within it, using an `int` would be a bad model for the real identifier. Notice that we are also creating a getter and a setter for `id`.

From here, we can create instances of `Student`. We can get and set the name and `id`.

```
> Student betsy = new Student()
> betsy.getName()
"Not-yet-named"
> betsy.setName("Betsy")
> betsy.getName()
"Betsy"
> betsy.setId(999)
> betsy.getId()
999

```

But, what happens if we try to create a student with a name?

```
> Student betsy=new Student("Betsy")
NoSuchMethodException: constructor Student(java.lang.String)

```

Why did this happen? We were able to create a new student object using `new Student()` but we didn't specify any constructors in the class `Student`. How then were we able to execute `new Student()`? If you don't provide any constructors, the Java compiler will create a constructor for you that doesn't take any parameters. This is called the *no-argument constructor*. But, once you add any constructors Java will not automatically create the no-argument one for you anymore.

If we print out `Betsy`, she'll tell us that she's a `Person`, not a `Student`. That's because the class `Student` inherits the `toString` method from `Person`. The method that gets called is the one in `Person`.

```
> betsy
Person named Betsy

```

Let's define the class `Student` with reasonable constructors and with a new `toString` method.

Example Java Code: **Class Student, with constructors and toString method** *Program Example #6*

```

public class Student extends Person {

    //////////////////// fields //////////////////////

    private int id;

    //////////////////// constructors //////////////////////

    public Student() {
        super(); //Call the parent's constructor
        id = -1;
    }

    public Student(String name) {
        super(name);
        id = -1;
    }

    //////////////////// methods //////////////////////

    public int getId() { return this.id;}

    public void setId(int theId) {this.id = theId;}

    public String toString() {
        return "Student named " + this.getName() +
            " with id " + this.id;
    }
}

```

An updated UML diagram for Person and Student shows that the Student class also has the `toString` method now so that it overrides the one in Person (Figure 2.5). Notice that we don't show the getter and setter methods in the UML class diagram. If they are not shown, they are implied to exist for each named field.

**How it works:** We create a default value for the `id` as `-1`. We also create an accessor, `getId()` and a mutator or modifier `setId()`, for manipulating the identification. Note that we still want to do whatever the Person constructors would do, so we call them with **super**. We added a `toString` method to print out the student's name and `id`.

```

> Student betsy = new Student("Betsy");
> System.out.println(betsy);
Student named Betsy with id -1

```

\* \* \*

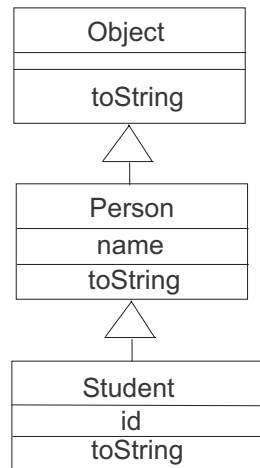


Figure 2.5: A modified UML class diagram for Person and Student

#### Debugging Tip: Make the call to `super` *first* in a subclass constructor

If a subclass constructor doesn't have a call to `super` as the first statement in the constructor, Java will automatically add a statement that calls the parent's constructor that takes no parameters using `super()`. So, we don't have to have the `super()` call in the constructors above, since it will be added for us if we don't include it, but we wanted to show that it will happen. If you want to call a different constructor than the one that takes no parameters in a subclass constructor make sure that it is the first statement in the subclass constructor.

#### Creating a `main()` method

How do we test our Java code? For those of us using DrJava<sup>1</sup>, it's easy—we can simply type code in the INTERACTIONS PANE. What if you don't *have* an INTERACTIONS PANE? Then, you can test code by providing a `main()` method. There can be at most one `main` method in a class. It is often used to try out the class.

A `main` method is declared as `public static void main(String[] args)`, which is a real mouthful. That means (in brief—there is a better explanation later in the book):

- **public** means that this method can be accessed by any other class.

<sup>1</sup>Or BlueJ, <http://www.bluej.org>

- **static** means that this method is known to the *class*, not just the instances (objects). We can execute it without any objects of the class having been created. This is required because when we first load a class there aren't any objects that have been created from the class.
- **void** means that this method does not return anything.
- `String[] args` means that this method *could* be executed by running it from a command line, and any words on that line (e.g., filenames or options) would be passed on to the main method as elements in an array of strings named `args` (for "arguments").

Program  
Example #7

Example Java Code: **main() method for Person**

```
public static void main(String[] args) {
    Person fred = new Person("Fred");
    Person barney = new Person("Barney");

    System.out.println("Fred is a " + fred);
    System.out.println("Barney is a " + barney);
}
```

**How it works:** Add the main method to the Person class and compile the class. You can then `run` the class (however your Java environment allows you to do this) or execute it from the command line and this will execute the main method. Two instances (objects) of class Person will be created, and then both are printed to the console (or Interactions Pane, for DrJava) using the `toString` method.

When we execute the main method by clicking the RUN button, we see:

```
Welcome to DrJava.
> java Person
Fred is a Person named Fred
Barney is a Person named Barney
```

## Exploring Inheritance

A subclass inherits all the public object methods in the superclass. If the subclass defines its own version of a public method in the superclass, then the subclass version will be executed instead of the version in the parent class. Let's add a method to class Person that allows instances of Person to be friendly and greet people.

\* \* \*

am  
ple #8

### Example Java Code: **greet method for Person**

```
public void greet() {
    System.out.println("Hi! I am " + this.name);
}
```

Executing this method looks like this:

```
> Person bruce = new Person("Bruce")
> bruce.greet()
Hi! I am Bruce
> Student kristal = new Student("Kristal")
> krista.greet()
Hi! I am Kristal
```

Let's imagine that we create a `greet` method for class `Student`, too. We might try something like this:

```
public void greet() {
    System.out.println("Hi! I'm " + this.name +
        " but I have to run to class...");
}
```

Unfortunately, that will generate an error when we try to compile `Student`.

```
1 error found:
File: C:\dsBook\java-source-final\Student.java [line: 26]
Error: name has private access in Person
```

That error occurred because `name` is a **private** variable. The subclass, `Student` can't directly access the variable using `this.name`. Only the class that defined it, `Person` can directly access it, since it is declared **private**. Instances of the class `Student` certainly *have* an `name` variable—they just can't access it directly. If we want to access the variable `name` in `Student`, we have to use the public accessor method `getName`.

### Example Java Code: **greet method for Student**

*Program  
Example #9*

```
public void greet() {
    System.out.println("Hi! I'm " + this.getName() +
        " but I have to run to class...");
}
```

\* \* \*

```

Now we get different results when we ask a Person and Student to greet().
> Person bruce = new Person("Bruce")
> bruce.greet()
Hi! I am Bruce
> Student krista = new Student("Krista")
> krista.greet()
Hi, I'm Krista, but I have to run to class...

```

It's worth thinking this through—what exactly happened when we executed `krista.greet()`?

- Krista knows that she is a Student—an instance of the class Student. Krista knows that if she can't greet, she can ask her parent to do it.
- She does know how to `greet()`, so she executes that method.
- But midway, “Uh-oh. I don't know how to `getName()`!” So Krista asks her parent (who might have asked her parent, and so on, as necessary), who does know how to `getName()`.

Let's try one more experiment:

```

> Person fred = new Student("Fred")
> Student mabel = new Person("Mabel")
ClassCastException: mabel

```

Why did the first statement work, but the second one generated an error? Variables in Java always have a particular *type*. They can refer to values that match that type. A Person variable `fred` can refer to a Person object. A variable of a class type can also refer to instances of any *subclasses*. Thus, Person variable `fred` can refer to an instance of class Student. However, a variable of a given type can *not* refer to objects of any *superclasses*. Thus, the Student variable `mabel` cannot refer to an instance of the class Person.

There are some subtle implications when referring to an object of one type in a variable of another type. Consider the following example:

```

> fred.greet()
Hi, I'm Fred, but I got to run to class...
> fred.setId(999)
Error: No 'setId' method in 'Person' with arguments: (int)
> ((Student) fred).setId(999)
> ((Student) fred).getId()
999

```

Our Person variable `fred` knows how to `greet()`—of course it does, since both Person and Student classes know how to greet. Note that `fred` executes the *Student* method `greet()`—that also makes sense, since `fred` refers to an instance of the class Student.

Here's the tricky part: `fred` can't `setId()`. You may be thinking, “But Fred's a Student! Students know how to set their id!” Yes, that's true. The

variable `fred`, though, is declared to be type `Person`. That means that Java is checking that `fred` is only asked to do things that a `Person` might be asked to do. A `Person` does not know how to set its `id`. We can tell Java, “It’s okay—`fred` contains a `Student`” by *casting*. When we say `((Student) fred)`, we are telling Java to treat `fred` like a `Student` and let it execute `Student` methods. Then it works.

## 2.4 Manipulating Pictures in Java

We have created Java classes to make it easy for you to create and manipulate pictures and sounds. These classes are in the `java-source` directory. You also have a `media-source` directory that contains files that contain picture and sound data. You can create a `Picture` object by passing it a full file name. You can get a *full file name* by using the `FileChooser` class and its’ method `pickAFile()`. The method `pickAFile()` is special in that it’s known to the class as well as to objects created from that class (*instances*). It’s called a **static** or *class method*. To access a class method, use: `ClassName.methodName()`.

```
> String file = FileChooser.pickAFile();
> System.out.println(file);
"C:\dsBook\media-source\beach-smaller.jpg"
```

In the array example in section 2.2, we used two class methods: `FileChooser.pickMediaPath` and `FileChooser.getMediaPath`. The method `pickMediaPath()` will let you pick a directory that holds your media (`media-source`) and save that directory path for later use. The method `getMediaPath(filename)` takes a filename, then returns the media directory pathname concatenated in front of it. So `FileChooser.getMediaPath("jungle.jpg")` actually returns the full file name of `"c:/dsbook/media-source/jungle.jpg"`. *You only need to use `pickMediaPath` once!* The pathname for the media gets stored in a file on your computer, so that all your code that uses `getMediaPath` will just work. This makes it easier to move your code around.

Recall that just declaring a variable to refer to a picture sets the value of that variable to `null` to show that it doesn’t refer to an object yet.

```
> Picture p;
> p
null
```

### Debugging Tip: Did you get an error?

If you got an error as soon as you typed `Picture p`; there are two main possibilities.

- All the Java files we provide you are in source form. You need to compile them to use them. Open `Picture.java` and click **COMPILE ALL**. If you get additional errors about classes not found, open those files and compile them, too.
- You might not have your **PREFERENCES** set up correctly. If Java can't find the `Picture` class, you can't use it. Be sure to add the `java-source` directory to your classpath.

### Debugging Tip: Semicolons or not?

In the DrJava Interactions Pane, you don't have to end your lines with a semicolon (;). If you don't, you're saying to DrJava "Evaluate this, and show me the result." If you do, you're saying "Treat this like a line of code, just as if it were in the Code Pane." Leaving it off is a useful debugging technique—it shows you what Java thinks that variable or expression means. But be careful—you *must* have semicolons in your Code Pane!

To make a new picture, we use the code (you might guess this one) `new Picture(fullFileName)`. Then we'll have the picture show itself by telling it (using dot notation) to `show()` (Figure 2.6).

```
> p = new Picture("c:/dsBook/media-source/beach-smaller.jpg");  
> p  
Picture, filename c:/dsBook/media-source/beach-smaller.jpg  
height 360 width 480  
> p.show()
```

The variable `p` in this example has the type `Picture`. That means that it can only refer to pictures or subclasses of `Picture`. The variable `p` can't refer to a `Sound` or `int`. We also can't re-declare `p`.

### Common Bug: One declaration per scope

Within a given *scope* (e.g., any set of curly braces, such as a single method, or the Interactions Pane in DrJava between compilations or after a reset), a variable can be declared once and only once. Another declaration with the same name is an error. You can change what the variable refers to as you might like after declaration, but you can only *declare* it once.

After the scope in which it was declared, the variable ceases to exist. So, if you declare a variable inside the curly braces of a **for** or **while** loop, it will not be available *after* the end curly brace.

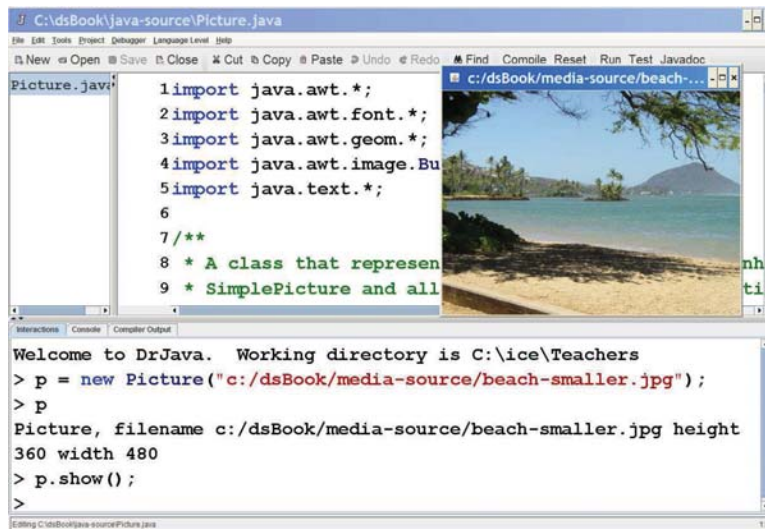


Figure 2.6: Showing a picture

\* \* \*

**Common Bug: Java may be hidden on Macintosh**

When you open windows or pop-up file choosers on a Macintosh, they will appear in a separate “Java” application. You may have to find it from the Dock to see it.

The downside of types is that, if you need a variable, you need to create it. In general, that’s not a big deal. In specific cases, it means that you have to plan ahead. When we modify pictures we will actually change the color of the individual *picture elements*. Picture elements are also called *pixels*. Let’s say that you want a variable to refer to a pixel (class `Pixel`) that you’re going to assign inside a loop to each pixel in a list of pixels. In that case, the declaration of the variable *should* be *before* the loop. If the declaration was inside the loop, you’d be re-creating the variable each time through the loop, rather than just changing the value of the variable.

To refer to an array of pixels, we use the notation `Pixel[]`. The square brackets are used in Java to index an array. In this notation, the open-close brackets means “an array of indeterminate size.” We can use the method `getPixels` to get a one-dimensional array of all the pixels in the picture.

Here’s an example of increasing the red for each pixel of a picture by doubling the original red value (Figure 2.7).

```
> Pixel[] pixelArray = p.getPixels();
```

```

> for (Pixel pixelObj : pixelArray) {
    pixelObj.setRed(pixelObj.getRed()*2);
}
> p.show()

```

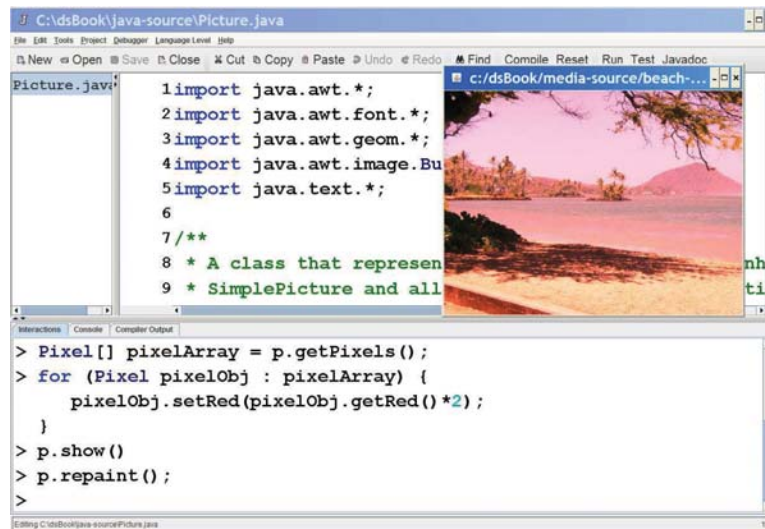


Figure 2.7: Doubling the amount of red in a picture

How would we put this process in a file, something that we could use for *any* picture? If we want *any* picture to be able to double the amount of red, we need to edit the class `Picture` in the file `Picture.java` and add a new method, maybe named `doubleRed`.

Here's what we would want to type in. The special variable **this** will represent the `Picture` instance that is being asked to double red. (In Python or Smalltalk, **this** is typically called `self`.)

*Program*  
*Example #10*

#### Example Java Code: **Method to double the red in a Picture**

```

/**
2  * Method to double the red in a picture.
3  */
4  public void doubleRed() {
5      // get the array of pixels for this picture
6      Pixel [] pixelArray = this.getPixels();
7
8      // loop through all the pixels in the array

```

```

10     for (Pixel pixelObj : pixelArray) {
12         // set the red at this pixel to twice the original value
13         pixelObj.setRed(pixelObj.getRed()*2);
14     }

```

#### How it works:

- The notation `/**` begins a Javadoc multi-line comment in Java – stuff that the compiler will ignore. The notation `*/` ends the comment. Javadoc is a utility that will read our Java source files and create html documentation from the Javadoc comments. Java also has two other type of comments that are ignored by the Javadoc utility. One is a single line comment using `//`. This type of comment can be anywhere on a line and comments out the rest of the line. The other is a multi-line comment using `/*` to start and `*/` to end. This comments out all lines between the start and end symbols.
- We have to declare methods just as we have to declare variables! The term **public** means that any class can use this method. Why would we want a method to be **private**? Sometimes you want a method to only be used inside the class and not available to other classes. The term **void** means “this is a method that doesn’t return anything—don’t expect the return value to have any particular type, then.”

Once we type this method into the bottom of class `Picture` (before the closing curly brace `)`, we can press the `COMPILE` button. If there are no errors, we can test our new method. When you compile your code, the objects and variables you had created in the Interactions Pane disappear. You’ll have to recreate the objects you want.

#### Making It Work Tip: The command history isn’t reset!

Though you lose the variables and objects after a compilation, the history of all commands you typed in DrJava is still there. Just hit up-arrow to get to previous commands, then hit return to execute them again.

You can see how this works in Figure 2.8.

```

> Picture p = new Picture(FileChooser.pickAFile());
> p.explore();
> p.doubleRed();
> p.explore();

```

The method `explore` will make a copy of the current picture and open it in a picture explorer. The picture explorer will let you see the red, green,

and blue values for any pixel in a picture.

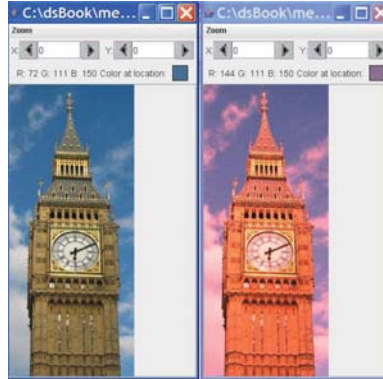


Figure 2.8: Doubling the amount of red using our `doubleRed` method on `bigben.jpg`

What would happen if the amount of red at a pixel is already near the maximum value of 255 and you try to double it? What do you think should happen? You can call the method `doubleRed` several times on a picture and use the explorer to check what actually happens.

Later on, we're going to want to have characters moving to the left or to the right. We'll probably only want to create one of these (left or right), then flip it for the other side. Let's create the method for doing that. Notice that this method returns a *new* picture, it doesn't modify the original one. Instead of being declared with a return type of `void`, the `flip` method is declared with a return type of `Picture`. This means that it must return a picture. At the bottom of the method, you'll see that it uses `return` to return the target picture that we created inside the method. We'll see later that that's pretty useful, to create a new image rather than change the target picture. (Figure 2.9).

*Program*  
*Example #11*

Example Java Code: **Method to flip an image**

```

2      /**
3       * Method to flip a picture
4       */
5      public Picture flip() {
6          // declare some local variables
7          Pixel currPixel = null;
8          Pixel targetPixel = null;
9          Picture target =
10             new Picture(this.getWidth(), this.getHeight());

```

```

10
11     /* loop through the picture with the source x starting at 0
12     * and the target x starting at the width minus one
13     */
14     for (int srcX = 0, trgX = getWidth()-1;
15          srcX < getWidth();
16          srcX++, trgX--) {
17         for (int srcY = 0, trgY = 0;
18              srcY < getHeight();
19              srcY++, trgY++) {
20
21             // get the current pixel
22             currPixel = this.getPixel(srcX,srcY);
23             targetPixel = target.getPixel(trgX, trgY);
24
25             // copy the color of currPixel into target
26             targetPixel.setColor(currPixel.getColor());
27         }
28     }
29     return target;
30 }

```

```

> Picture p = new Picture(FileChooser.pickAFile());
> p
Picture, filename c:\dsBook\media-source\mLeft1.jpg height 264 width 97
> p.show();
> Picture flipP = p.flip();
> flipP.show();

```

### Common Bug: Width is the size, not the index

Why did we subtract one from `getWidth()` (which defaults to `this.getWidth()`) to set the target X coordinate (`trgX`)? `getWidth()` returns the *number of pixels* across the picture. But the last valid *index* in the row is one less than that, because Java starts all arrays at index *zero*. The method `getWidth` returns the number of pixels not the last index.

## 2.5 Exploring Sound in Java

We can create sounds in much the same way we created pictures.

```

> Sound s = new Sound(FileChooser.pickAFile());
> s.play();
> s.explore();

```

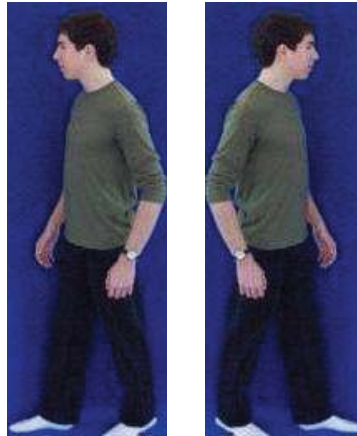


Figure 2.9: Flipping our guy—original (left) and flipped (right)

**How it works:** Just as with pictures, we can create sounds when we declare them. `FileChooser` is a class that knows how to `pickAFile()`. That method puts up a file picker, then returns a string (or `null`, if the user hits CANCEL). Instances of the class `Sound` know how to `play()`.

But what if we get it wrong?

```
> s.play()
> s.show()
Error: No 'show' method in 'Sound'
> Picture.play()
Error: No 'play' method in 'Picture'
> anotherpicture.play()
Error: Undefined class 'anotherpicture'
```

You can't ask a `Sound` object to `show()`—it doesn't know how to do that. `Picture` (the class) doesn't know how to `play()` nor how to `show()`—it's the *instances* (objects of that type or class) that know how to `show()`. The point of this example isn't to show you Java barking error messages, but to show you that there is no bite there. Type the wrong object name? Oh well—try again.

## 2.6 Exploring Music in Java

We will be working a lot with *MIDI* in this class. *MIDI* is a standard representation of musical information. It doesn't record sound. It records notes—when they're pressed, when they're released, how hard they're pressed, and what instrument is being played.

To use *MIDI*, we have to **import** some additional libraries. We're go-

ing to be using *JMusic* which is a wonderful Java music library that is excellent for manipulating MIDI.

```
> import jm.util.*;
> import jm.music.data.*;
> Note n1;
> n1 = new Note(60,0.5);
> // Create an eighth note at C octave 4
```

**How it works:** First, there are a couple of **import** statements which allow us to use short names for classes in *JMusic*. *JMusic* classes are in packages and the full name for a class in a package is the package name followed by a period followed by the class name. But, if we import all the classes in a package we can just use the class name to refer to a class in a package. *Note* is the name of the class that represents a musical note object. We're declaring a note variable named *n1*. We then create a *Note* instance (object). We don't need a filename—we're not reading a JPEG or WAV file. Instead, we simply need to know *which* note and for what duration (0.5 is an eighth note). That last line looks surprisingly like English, because it is. Any line starting with *//* is considered a *comment* and is ignored by Java. Table 2.1 summarizes the relationships between note numbers and more traditional keys and octaves.

Octave #	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Table 2.1: MIDI notes

But this isn't actually enough to play our note yet. A note isn't music, at least not to *JMusic*.

```
> Note n2=new Note(64,1.0);
> View.notate(n1);
Error: No 'notate' method in 'jm.util.View' with arguments:
(jm.music.data.Note)
```

```

> Phrase phr = new Phrase();
> phr.addNote(n1);
> phr.addNote(n2);
> View.notate(phr);
-- Constructing MIDI file from 'Untitled Score' ... Playing with
JavaSound ... Completed MIDI playback -----

```



Figure 2.10: Just two notes

**How it works:** You just saw that we can't `notate()` a single note. We can, however, create a phrase that can take two notes Figure 2.10. A Phrase in JMusic knows how to add a note using the method `addNote()`. The View class knows how to display a phrase of music in standard Western music notation using the method `notate()`. Notice that we didn't have to create a View object. We just called the `notate` method on the class since it is a class method. From the displayed window, we can actually play our music, change parameters (like the speed at which it plays), and shift instruments (e.g., to accordion or wind chimes or steel drums). We'll do more with the display window later.

JMusic is a terrific example of using objects to *model*. JMusic is really modeling music. We can break down the differences between Note instances and Phrase instances in terms of what they *know* and what they *do*.

- Note objects have tones and durations.
- Musical Phrase objects are collections of notes.
- The View class can display a musical phrase.

	<b>What instances of this class <i>know</i></b>	<b>What instances of this class <i>do</i></b>
Note	A musical pitch and its duration.	(Nothing we've seen yet.)
Phrase	Notes in the phrase.	<code>addNote(aNote)</code>

The entire collection of objects in a JMusic Score is a model of the next levels up of Western music (Figure 2.11). We can associate instruments

with Part instances. We could associate them with Phrase instances, but that doesn't mesh with how real worlds part work. Can you imagine the First Violin swapping instruments for a saxophone mid-way through a piece? Part instances are collected into a Score. Scores have a tempo. Rarely do different real world parts have different tempos.

Note that a Phrase instance can have any number of Notes, a Part can have any number Phrases, and a Score can have any number of Parts. How might that be implemented? Almost certainly, none of these objects contains a pre-sized Array. These objects are using dynamic data structures that can expand its size.

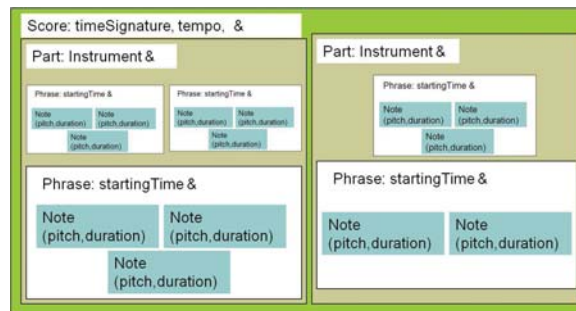


Figure 2.11: Structure of a score in terms of JMusic objects

## Exercises

- Define the following:
  - inheritance
  - polymorphism
  - overloading
  - overriding
  - association
  - subclass
  - superclass
  - instance
  - field
  - method
- Declare an array of String objects and set the array elements to 5 different strings.

3. Declare a variable that will represent if you have eaten. Declare a variable that will represent the score in a soccer game. Declare a variable that will represent the price of an item.
4. Declare more than one variable that refers to the same object.
5. Try to declare two variables with the same name in the same scope. What happens?
6. We have been using the type `int` to represent integers. There are other types that you can use to represent integers in Java. What are the other types and what are the differences between them?
7. Try to divide 1 by 3 in the interactions pane. What result do you get? Is this what you expect? Why do you get this result?
8. Try to divide 1.0 by 3.0 in the interactions pane. what result do you get? Is this what you expect? Why do you get this result?
9. Look up Unicode on the web? What is it and why does Java use it to represent characters?
10. Look up 2's compliment on the web? What is it and how does it represent negative numbers?
11. Look up IEEE 754 floating point format on the web? What it is and how does it represent floating point numbers?
12. Create a constructor for Student that takes a name and an id.
13. Add a private field `eMail` to the Person class. This field will be used to store a person's e-mail address. Add accessor and modifier methods for this private field.
14. What happens when you try to directly modify a private field in a child class using dot-notation to access the field? Why does this happen?
15. Create a class that models a teacher. The Teacher class should also inherit from the Person class. Are there any fields we should add to the Teacher class?
16. Create a class that models a credit card. It should contain fields for the person's name, the credit card number, the expiration date, and the security code. Be sure to include constructors, accessors, and modifier methods.
17. Create a class that models a car. The Car class should have fields for color, manufacturer, model, and year. Be sure to include constructors, accessors, and modifier methods.

18. Create a class that models a class session. Each class session should have an array of students in that class session. Each class session should have a teacher.
19. Add a method to the `Picture` class that sets the red and blue values for each pixel to zero. You can call this `keepOnlyGreen`. Do you think that you will still be able to tell what is in the picture?
20. Add a method `zeroBlue` to the `Picture` class that sets the blue value to zero for every pixel in the picture.



# 3 Methods in Java: Manipulating Pictures

## Chapter Learning Objectives

To make the wildebeests charge over the ridge or the villagers move around in the town square, we need to be able to manipulate pictures. We manipulate pictures using *methods*, the bits of behavior that classes store for use on their objects. That's the focus of this chapter.

**The computer science goals for this chapter are:**

- To become more familiar with Java syntax and control statements.
- To create a variety of different kinds of methods, including those that return values.
- To recognize Javadoc comments.
- To chain method calls for compact, powerful expressions.
- To start our discussion of data structures with arrays.

**The media learning goals for this chapter are:**

- To extend what we can do with pictures.
- To combine methods for powerful picture manipulation.

## 3.1 Reviewing Java Basics

### Assignment

As we saw in the last chapter, assignments come in form of

```
TYPE NAME = EXPRESSION;
```

or simply (if the variable has already been declared)

```
NAME = EXPRESSION;
```

As mentioned, we can't declare variables twice in the same scope, and you can't use a variable of one type (or class) with an expression that results in an incompatible type. You can't assign a String object to an `int` variable, for example,

**Making It Work Tip: DrJava will declare variables for you—maybe not a good thing**

If `sound` is an undeclared variable, DrJava will actually allow you to execute `sound = new Sound("D:/myfile.wav");` in the interactions pane. DrJava is smart enough to figure out that you must intend that `sound` to be of type `Sound`. My suggestion: Don't rely on this. Be explicit in your type declarations. It will be too easy to forget to declare the variables when you're in Java Code Pane and you must declare them there before you use them.

There are rules about *Java programming style* that you should know about. These aren't rules that, if broken, will result in a compiler error (usually). These are rules about how you write your code so that other Java programmers will understand what you're doing. We might call them *discourse rules*—they're the standard style or ways of talking that Java programmers use.

- Always start class names with a capital letter, like `String`.
- Never start variables names, field names, or method names with a capital letter! These names should start with a lowercase word, like `sound`.
- You should capitalize the first letter of each additional word in a name to make it easier to read, like `pixelArray`.

All Java statements end with a semicolon. You can insert as many spaces or returns (press the ENTER key) as you want in an expression—it's the semicolon that Java uses to indicate the end of the line. Indentation doesn't matter at all in Java, unlike in other languages like Python. You can have no indentation at all in Java! Of course, no one, including you, will be able to make out what's going on in your program if you use no indentation at all. You probably should indent as we do here, where the body of a loop is indented deeper than the loop statement itself. DrJava will take care of that for you to make it easier to read, as will some other Java Integrated Development Environments (*IDEs*).

What goes in an expression? We can use `+`, `-`, `*` and `/` exactly as you used them in whatever your first programming language was. An expression that you've seen several times already is `new ClassName()`, sometimes with inputs like `new Picture("C:/mypicture.jpg")`. You may have already noted that sometimes you create a new class with inputs, and some-

times you don't. Whether or not you need inputs depends on the *constructors* for the given class—constructors initialize the new object and set the field values to any passed inputs. For example, when we created a new Note with a pitch and a duration, we passed the specification of the pitch and duration in as input to the class Note and they were used to initialize the new object.

Java has a handful of shortcuts that you will see frequently. Because the phrase `x = x + 1` (where `x` could be any integer variable) occurs so often, we can abbreviate it `x++`. Because the phrase `y=y-1` occurs so often, it can be abbreviated as `y--`. There's a general form, too. The phrase `x = x + y` can be shortened to `x += y`. There are similar abbreviations `x *= y`, `x /= y`, and `x -= y`.

## Arrays

An array is a homogenous (all items are of the same type) linear collection of objects which are compacted together in memory. An array of integers, then, is a whole bunch of numbers (each without a decimal place), one right after another in memory. Being all scrunched together in memory is about as efficient in terms of memory space as they can be, and they can be accessed very quickly—going from one to the other is like leaving your house and going to the house next door.

An array is declared with square brackets `[]`. The square brackets can come before or after the variable name in a declaration, so both of the below are correct Java statements (though clearly you can't use both in the same scope!).

```
Pixel[] myPixels;  
Pixel myPixels[];
```

It is important to note that neither of the ways to declare an array actually creates an array. They both just declare a reference to an array of Pixel objects. The value of `myPixels` will be automatically set to **null** as shown below.

```
> Pixel[] myPixels;  
> System.out.println(myPixels);  
null
```

To access an array, we'll use square brackets again, e.g., `myPixels[0]`, which gets the first element in the array. Java begins numbering its indices at zero.

## Conditionals

You've seen already that conditionals look like this:

```
if (LOGICAL-EXPRESSION)
    then-statement;
```

As you would expect, the logical expression can be made up of the same logical operators you've used before: `<`, `>`, `<=`, `>=`, `==`. Note that `==` is the test for equivalence—it is not the assignment operator `=`. Depending on other languages you've learned, you may have used the words *and* and *or* for chaining together logical statements, but not in Java. In Java, a logical *and* is `&&`. A logical *or* is `||`.

The *then-statement* part can be one of two things. It could just be a simple statement ending in a semi-colon (e.g., `pixel.setRed(0)`). Or it could be any number of statements (each separated by semi-colons) inside of curly braces, like this:

```
if (pixel.getRed() < 25) {
    pixel.setRed(0);
    pixel.setBlue(120);
}
```

Do you need a semicolon after the last curly brace? No, you don't have to, but if you do, it's not wrong. All of the below are correct conditionals in Java.

```
if (thisColor == myColor)
    setColor(thisPixel, newColor);
if (thisColor == myColor) {
    setColor(thisPixel, newColor);
}
if (thisColor == myColor) {
    x = 12;
    setColor(thisPixel, newColor);
}
```

We call a set of open and close curly braces a *block*. A block is a single statement to Java. All the statements in the block together are considered just one statement. Thus, we can think of a Java statement ending in a semicolon or a right curly brace (like an English sentence can end in `.` or `!` or `?`).

After the block for the *then* part (the part that gets executed "if" the logical expression is true, as in "if this, then that") of the `if`, you can have the keyword `else`. The `else` keyword can be followed with another statement (or another block of statements) that will be executed if the logical expression is *false*. You can't use the `else` statement if you end the *then* block with a semicolon though (like in the last `if` in the example above). Java gets confused if you do that, and thinks that you're trying to have an `else` without an `if`.

**Iteration: For each, While and For**

As of Java 1.5 (also called Java 5) you can use a for-each loop to loop through the elements of an array or collection and repeat a statement or block of statements.

```
for (TYPE NAME : ARRAY)
    STATEMENT
```

We used the for-each statement in the last chapter to loop through all pixels of a picture.

**Example Java Code: Method to increase red in Picture**

*Program  
Example #12*

```

2  /**
   * Method to double the red in a picture.
   */
4  public void doubleRed() {
    Pixel [] pixelArray = this.getPixels();
6    for (Pixel pixelObj : pixelArray) {
      pixelObj.setRed(pixelObj.getRed()*2);
8    }
  }
}
```

**A while loop looks like an if:**

```
while (LOGICAL-EXPRESSION)
    statement;
```

But they're not at all similar. An **if** tests the expression *once* then executes the *then* statement if the expression was true. A **while** test the expression, and if true, executes the statement—then test the expression again, and again executes the statement, and repeats until the expression is no longer true. That is, a **while** statement *iterates*.

We can use a **while** for addressing all the pixels in an image and setting all the red values to zero. We just have to walk through the elements of the array ourselves.

```
> p
Picture, filename D:/cs1316/MediaSources/Swan.jpg height 360 width
480
> Pixel [] myPixels = p.getPixels();
> int index = 0;
> while (index < myPixels.length) {
    myPixels[index].setRed(0);
    index++;
}
```

```
}

```

**How it works:** Notice the reference to `myPixels.length` above. This is the standard way of getting an array's length. The expression `.length` isn't referring to a method. Instead it's referring to a public *instance variable* or *field*. Every array has a field that stores its length. Each length is an instance variable unique to that instance. It's all the same name, but it's the right value for each array.

Recall that the **for** loop is unusual. It has three parts: *initialization* (something to be done before the loop starts), *continuing condition* (something to test at the top of each loop), and *change area* (what to do after executing the body of the loop). It's actually the same structure as in the programming languages C and C++. We can use a **for** loop to count from one value to another, just as you might use a **for** loop in Basic or Python. But you can also use the Java **for** loop to do a lot more, like walk through both the `x` and `y` values at once.

Here's the same example as the **while** loop above, but with a **for** loop:

```
> for (int i=0; i < myPixels.length ; i++) {
    myPixels[i].setRed(0);
}
```

**How it works:** Our *initialization* part is declaring an integer (**int**) variable `i` and setting it equal to zero. Notice that `i` will *ONLY* exist within the **for** loop. On the line afterward, `i` won't exist—Java will complain about an undeclared variable. The *continuing condition* is `i < myPixels.length`. We keep going until `i` is equal to the length, and we *don't* execute when `i` is equal to the length. The *change area* is `i++`—increment `i` by one. What this does is to make `i` take on every value from 0 to `myPixels.length-1` (minus one because we stop when `i IS` the length), and execute the body of the loop—which sets red of the pixel at `i` equal to zero.

## 3.2 Java is about Classes and Methods

In Java, nearly everything is an object, except for the primitive types (`int`, `double`, `char`, and `boolean`). Objects *know* things and *know how to do* things. It's the objects that do and know things—there aren't globally accessible functions like there are in many other languages like Python, C, or Smalltalk.

Programming in Java (and object-oriented languages, in general) is about defining what these objects know and what they know how to do. Each object belongs to a *class*. The class defines what the object knows (its *instance variables* or *fields*) and what it knows how to do (its *methods*).

For the `Picture` object, there is a file named `Picture.java` that defines the fields and methods that all `Picture` objects know (Figure 3.1). That file starts out with the line **public class** `Picture`. That starts out the definition of the class `Picture`. Everything inside the open curly brace and matching

close curly brace at the end of the file is part of the definition of the class `Picture`.

Typically, we define at the top of the file the instance variables (fields) that *all* objects of that class know. After that we define the constructors that are used to initialize the fields in new objects. Next we define the methods. The fields, constructors, and methods must be inside the open and close curly braces for the class definition. Each object (e.g., each `Picture` instance) has the same instance variables, but different values for those variables—e.g., each picture has a filename where it read its file from, but the filenames for different picture objects will be different. All picture objects know the same methods—they *know how to do* the same things.

Picture.java

```
public class Picture {  
  
    //////////////// fields //////////////////////  
  
    //////////////// constructors //////////////////////  
  
    //////////////// methods //////////////////////  
  
}
```

Figure 3.1: Structure of the `Picture` class defined in `Picture.java`

#### Debugging Tip: You change `Picture.java`

There should be one and only one `Picture.java` file. This means that you *have to* modify the file that we give you. If you rename it (say, `Picture-v2.java`), Java will just complain about the filename being incorrect. If you save `Picture.java` somewhere else, Java will get confused about two versions. Save a backup copy somewhere, and trust that it will be okay—you won't damage the file too severely.

So what's this **public** statement about the class `Picture`? You might be wondering if there are other options, like `discreet` or `celebrity`. The statement **public** means that the class `Picture` can be used by any other class. This is also called public *visibility*, as in who can see the class. In general,

every field or method has a visibility which can be **public**, or **protected**, or **private**, or if no visibility is specified the default is **package** visibility.

- **public** visibility means that the class, field, or method is accessible by anyone. If there was a class with **public** fields, any other object could read those fields or change the values in them. Is that a good thing? Think about it—if objects represent (model) the real world, can you read any value in the world or change it? Not usually.
- **private** visibility means that the field or method can *only* be accessed by the code in the class containing that field or method. That’s probably the best option for fields. Some methods might be **private**, but probably most are **public**.
- **protected** visibility is a middle ground that is only really useful for inherited methods. It means that the field or method is accessible by any class or its subclass—or any class belonging to the same *package*. “Package?” you say. “I haven’t seen anything about packages!” Exactly—and if you don’t deal with packages, **protected** data and methods are essentially **public**. Makes sense? Not to us either.
- **package** visibility means that the class is visible to classes in the same package. Again if you aren’t using packages you can ignore package visibility.

### Pictures are about arrays and pixels

A picture has a two-dimensional array of pixels. An array is typically one-dimensional in many programming languages—there’s just a collection of values, one right after the other. In a one-dimensional array, each element has a number associated with it called a *index variable*—think of it as the mailbox address for each element. A two-dimensional array is called a *matrix*—it has both height and width. In Java two-dimensional arrays are actually arrays of arrays. We usually number the columns and the rows. With pictures, the upper left hand corner is row number 0 (which we will refer to as the *y* index) and column number 0 (which we will refer to as the *x* index). The *y* values increase going down, and the *x* values increase going to the right.

Each unique pair of a column index and row index of a picture contains a *pixel*. A pixel is a picture element—a small dot of color in the picture or on the screen. Each dot is actually made up of a red component, a green component, and a blue component. Each of those components can have a value between 0 and 255. A red value of 0 is no red at all, and a red value of 255 has the maximum amount of red. All the colors that we can make in a picture are made up of combinations of red, green, and blue values, and each of those pixels sits at a specific  $(x, y)$  location in the picture.

### A method for decreasing red

Let's explore a method in the class `Picture` to see how this all works. Here's the method to decrease the red in a picture, which would be inserted in `Picture.java` inside the curly braces defining the class.

#### Example Java Code: `decreaseRed` in a `Picture`

*Program  
Example #13*

```

2  /**
   * Method to decrease the red by half in the current picture
   */
4  public void decreaseRed() {
6      Pixel pixel = null; // the current pixel
       int redValue = 0;   // the amount of red
8
       // get the array of pixels for this picture object
10     Pixel[] pixels = this.getPixels();
12
       // start the index at 0
       int index = 0;
14
       // loop while the index is less than the length of the pixels array
16     while (index < pixels.length) {
18
         // get the current pixel at this index
         pixel = pixels[index];
20         // get the red value at the pixel
         redValue = pixel.getRed();
22         // set the red value to half what it was
         redValue = (int) (redValue * 0.5);
24         // set the red for this pixel to the new value
         pixel.setRed(redValue);
26         // increment the index
         index++;
28     }
}

```

We can use this method like this:

```

> Picture myPic = new Picture("C:/dsBook/media-source/Barbara.jpg");
> myPic.show(); // Show the picture
> myPic.decreaseRed();
> myPic.repaint(); // Update the picture to show the changes

```

The first line creates a picture (`new Picture...`), declares a variable `myPic` to be a `Picture`, and assigns `myPic` to the new picture (the value at the

variable will be a reference to the picture). We then show the picture to get it to appear on the screen. The third line *calls* (or *invokes*) the method `decreaseRed` on the picture referred to by `myPic`. The fourth line, `myPic.repaint()`, tells the picture to update itself on the screen. The method `decreaseRed` changes the pixels within memory, but `repaint` tells the window on the screen to update from memory.

**How it works:** Note that this method is declared as returning **void**—that means that this method doesn’t return anything. It simply changes the object that it has been invoked upon. It’s **public** so anyone (any object) can invoke it.

At the beginning of the method, we typically declare the variables that we will be using. We will be using a variable `pixel` to hold each and every pixel in the picture. It’s okay to have a variable `pixel` whose class is `Pixel`—Java is case sensitive so it can figure out the variable names from the case names. We will use a variable named `redValue` to store the red value for each pixel before we change it (decrease it by 50%). It will be an integer (no decimal part) number, so we declare it to be an **int**.

#### **Making It Work Tip: Give local variables values as you declare them**

It’s considered good practice to give initial values to the local variables as you declare them. That way you know what is in there when you start, because you put it in there. Java does not give default values to local variables. Java does assign default initial values to all fields. Fields of type **int** or **double** have a default initial value of 0. Fields of type **boolean** have an initial value of **false**. Fields of any object type have a default initial value of **null**.

We give `redValue` an initial value of zero. We give `pixel` an initial value of **null**. **null** is the value that says, “This variable doesn’t refer to any object right now’.

The next line in `decreaseRed` is the statement that both declares the array of pixels and assigns the variable to refer to the array of pixels. The array that we assign it to is what the method `getPixels()` returns. (That’s the method `getPixels` which takes no inputs, but we still have to type `()` to tell Java that we want to call the method.) `getPixels` is a really useful method that returns all those pixels that are in the picture, but in linear, single-dimension array. It converts them from the matrix form (array of arrays) to a one-dimensional array form, to make them easier to process. Of course, as a one-dimensional array, we lose the ability to know what row or column a pixel is in, but if we’re doing the same thing to all pixels, we really don’t care.

Notice that the object that we invoke `getPixels()` on is **this**. What’s **this**? The object that we invoked `decreaseRed` on. In our example, we are calling

decreaseRed on the picture referred to by myPic (barbara.jpg). So **this** is the picture created from the file barbara.jpg.

We are going to use a variable named index to keep track of which pixel we are currently working on in this method. The first index in any array is 0, so we start out index as 0. We next use a **while** loop to process each pixel in the picture. We want to keep going as long as index is less than the number of pixels in the array. The number of elements in the array pixels is pixels.length.

length is not a method—it’s a *field* or an *instance variable*. It’s a value, not a behavior. We access it to get the number of elements in an array. Every array has a public field length.

**Common Bug: The maximum index is  $length - 1$**

A common mistake when working with arrays is to make the index go while it is less than or equal to *length*. The length is the *number* of elements in the array. The index numbers (the addresses on the array elements) start at 0, so the maximum index value is  $length - 1$ . If you try to access the element at index *length* you will get an error that says that you have an `OutOfBoundsException`—you’ve gone beyond the bounds of the array.

The body of the **while** loop in decreaseRed will execute repeatedly as long as index is less than *pixels.length*. Within the loop, we:

- Get the pixel at address index in the array pixels and make variable pixel refer to that pixel.
- Get the redness out of the pixel in variable pixel by calling the method getRed() and store it in redValue.
- We make redValue 50% smaller by setting it to 0.5 times itself. Notice that multiplying redValue by 0.5 could result in a value with a decimal point (think about odd values). But redValue can only be an integer, a value with no decimal point. So, we have to force the return value into an integer. To do that we *cast* the value into an integer, i.e., **int**. By saying “(**int**)” before the value we are casting (“(redValue \* 0.5)”), we turn it into an integer. There’s no rounding involved—any decimal part simply gets hacked off.
- We then store the new redValue back into the pixel with setRed(redValue). That is, we invoke the method setRed on the pixel in the variable pixel with the input redValue: pixel.setRed(redValue);
- Finally, we increment index with index++;. That makes index point toward the next value in the array, all set for the test at the top of the **while** and the next iteration through the body of the array.

**Method with an input**

What if we wanted to decrease red by an *amount*, not always 50%? We could do that by calling `decreaseRed` with an input value. Now, the code we just walked through doesn't take an input. Here's a method that does.

*Program  
Example #14*

Example Java Code: **decreaseRed with an input**

```
/**
 * Method to decrease the red by an amount
 * @param amount the amount to change the red by
 */
public void decreaseRed(double amount) {

    Pixel[] pixels = this.getPixels();
    Pixel p = null;
    int value = 0;

    // loop through all the pixels
    for (int i = 0; i < pixels.length; i++) {

        // get the current pixel
        p = pixels[i];
        // get the value
        value = p.getRed();
        // set the red value the passed amount time what it was
        p.setRed((int) (value * amount));
    }
}
```

**How it works:** This version of `decreaseRed` has something within the parentheses after the method name—it's the amount to multiply each pixel value by. We have to tell Java the type of the input value. It's the type **double**, meaning that it's a double-precision value, i.e., it can have a decimal point.

In this method, we use a **for** loop. A **for** loop has three parts to it:

- An initialization part, a part that occurs before the loop begins. Here, that's `int i = 0`. The semi-colons indicate the end of a part.
- A test part—what has to be true for the loop to continue. Here, it's that we have more pixels left, i.e., `i < pixels.length`.
- A change part, something to change each time through the loop. Here, it's `i++`, which increments the value in variable `i` by 1.

The rest of this loop is much the same as the other. *It's perfectly okay in Java to have both versions of decreaseRed in the Picture class at once.* Java can tell the difference between the version that takes no inputs and the one that takes one numeric input. We call the name, the number of inputs, and the types of the inputs as the *method signature*. As long as two methods have different method signatures, it's okay for them to both have the same name.

Notice the odd comment at the start of the method, the one with the @param notation in it. That is a specialized form of comment that is used to produce HTML documentation for the class using a utility called *Javadoc*. The Javadoc produced web pages for the classes provided with this book are in the doc folder inside the java-source folder. These Web pages explain all the methods, their inputs, how they're used, and so on (Figure 3.2). We sometimes refer to this information as the *API* or *Application Program Interface*. The content comes from these specialized comments in the Java files.

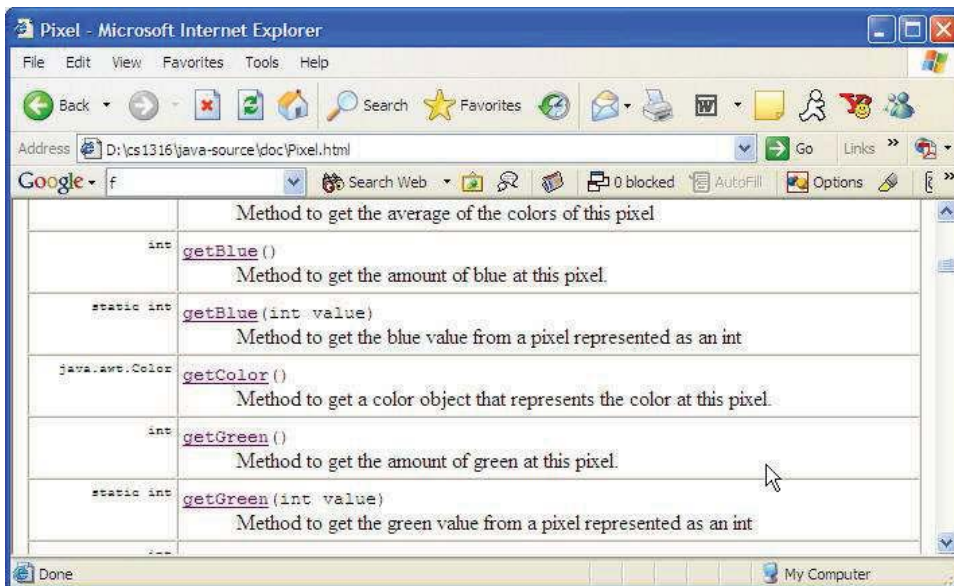


Figure 3.2: Part of the Javadoc page for the Pixel class

Now, if you look at the class `Picture`, you may be surprised to see that it doesn't know very much at all. Certainly, important methods like `show` and `repaint` are missing. Where are they? If you open the class `Picture` (in the file `Picture.java`), you'll see that it says:

```
public class Picture extends SimplePicture
```

That means that some of what the class `Picture` understands is actually defined in the class `SimplePicture`. Class `Picture` *extends* class `SimplePicture`. `Picture` is a subclass of `SimplePicture`, which means that class `Picture` *inherits* everything that `SimplePicture` has and knows. It's `SimplePicture` that actually knows about pixels and how pictures are stored, and it's `SimplePicture` that knows how to show and repaint pictures. `Picture` inherits all of that by being a *subclass* of `SimplePicture`.

Why do that? Why make `Picture` so relatively dumb? There are lots of reasons for using inheritance. The one we're using here is *information hiding*. Open up `SimplePicture.java` and take a peek at it. It's pretty technical and sophisticated code, filled with `BufferedImage` objects and references to `Graphics` contexts. We *want* you to edit the `Picture` class, to change methods and add new methods. We want that code to be understandable, so we hide the stuff that is hard to understand in `SimplePicture`.

### 3.3 Methods that return something: Compositing images

If we're going to make wildebeests or villagers, we need some way of getting those images onto a frame. Here are some methods that can do it. Along the way, we will create methods that return new pictures—a very useful feature for creating more complex pictures.

Program  
Example #15

Example Java Code: **Method to compose this picture into a target**

```

2      /**
3       * Method to compose (copy) this picture onto a target picture
4       * at a given point.
5       * @param target the picture onto which we copy this picture
6       * @param targetX target X position to start at
7       * @param targetY target Y position to start at
8       */
9      public void compose(Picture target, int targetX, int targetY) {
10         Pixel currPixel = null;
11         Pixel newPixel = null;
12
13         // loop through the columns
14         for (int srcX=0, trgX = targetX; srcX < this.getWidth();
15             srcX++, trgX++) {
16
17             // loop through the rows
18             for (int srcY=0, trgY=targetY; srcY < this.getHeight();
19                 srcY++, trgY++) {
20

```

```

22     // get the current pixel
    currPixel = this.getPixel(srcX,srcY);

24     /* copy the color of currPixel into target,
    * but only if it'll fit.
26     */
    if (trgX < target.getWidth() && trgY < target.getHeight()) {
28         newPixel = target.getPixel(trgX,trgY);
        newPixel.setColor(currPixel.getColor());
30     }
    }
32 }
}

```

We have been using `FileChooser.pickAFile()` to get the full pathname for a picture file or specifying the full pathname ourselves, but we can simply set the path to the directory that has the media (`media-source`) once and then use `FileChooser.getMediaPath("name.jpg")` which will add the saved media directory path in front of `name.jpg`. You can use `FileChooser.pickMediaPath()` in the INTERACTIONS PANE to pick the directory that has the media in it (`media-source`) or use `FileChooser.setMediaPath("directory/")` to set it. If you use `FileChooser.setMediaPath("directory/")` be sure to either double all backslash characters (`\\`) or use forward slashes (`/`). You do not have to set the media path again unless you change the location of the directory that has the media.

```

> FileChooser.pickMediaPath()
The media directory is now C:\dsBook\media-source/
> String file = FileChooser.getMediaPath("mLeft1.jpg")
> System.out.println(file)
"C:\dsBook\media-source\mLeft1.jpg"

```

Using the `compose` method defined above, we can compose (copy) a guy into the jungle like this (Figure 3.3).

```

> Picture p = new Picture(FileChooser.getMediaPath("mLeft1.jpg"));
> Picture jungle =
    new Picture(FileChooser.getMediaPath("jungle.jpg"));
> p.compose(jungle, 65, 200);
> jungle.show();
> jungle.write(FileChooser.getMediaPath("jungle-with-guy.jpg"));

```

**How it works:** Basically what happens in this method is that we copy the colors out of the source picture, **this**, and set the pixels in the target to those colors. That makes **this** picture (the one you invoked the method on) appear in the target picture.



Figure 3.3: Composing a guy into the jungle

The `compose` method takes three inputs. The first one is a picture onto which the **this** picture (the one that the method is being invoked upon) will be composed. Think of the input as a canvas onto which we paint this picture. The other two inputs are the  $x$  and  $y$  position where we start painting the picture—the variables `targetX` and `targetY` are integers that define where the upper left hand corner of this picture appears in the target picture.

We don't have the luxury of using `getPixels` this time. We need to know which rows and columns are which, so that we make sure that we copy them all into the right places. We don't want this picture (our source) to show up as one long line of pixels—we want the rows and columns in the source to show up as rows and columns in the target.

We are going to need two **for** loops. One is going to take care of the  $x$  indices, and the other will take care of the  $y$  indices. We use two variables for keeping track of the current pixel in **this** picture, our source. Those variables are named `srcX` and `srcY`. We use two other variables for keeping track of the current location in the target, `trgX` and `trgY`. The trick to a composition is to always increment `srcX` and `trgX` together (so that we're talking about columns in the source and the target at the same time), and `srcY` and `trgY` together (so that the rows are also in synch). You don't want to start a new row in the source but not the target, else the picture won't look right when composed.

To keep them in synch, we use a **for** loop where we move a couple of expressions in each part. Let's look at the first one in detail.

```
for (int srcX=0, trgX = targetX; srcX < this.getWidth();
      srcX++, trgX++)
```

- In the initialization part, we declare `srcX` and set it equal to zero, then declare `trgX` and have it start out as the input `targetX`. Notice that declaring variables here is *the same as* (for the **for** loop) declaring them inside the curly braces of the **for** loop's block. This means that

those variables *only* exist in this block—you can't access them after the class ends.

- In the testing part, we keep going as long as we have more columns of pixels to process in the source—that is, as long as `srcX` is less than the maximum width of this picture, `this.getWidth()`.
- In the change part, we increment `srcX` and `trgX` together.

#### Common Bug: Don't try to change the input variables

You might be wondering why we copied `targetX` into `trgX` in the `compose` method. While it's perfectly okay to use methods on input objects (as we do in `compose()` when we get pixels from the target), and maybe change the object that way, don't try to add or subtract the values passed in. It's complicated why it doesn't work, or how it does work in some ways. It's best just to use them as variables you can *read* and *call methods on*, but not *change*.

The body of the loop essentially gets the pixel from the source picture, gets the pixel from the target picture, and sets the color of the target pixel to the color of the source pixel. There is one other interesting statement to look at:

```
if (trgX < target.getWidth() && trgY < target.getHeight())
```

What happens if you have a really wide source picture and you try to compose it at the far right edge of the target? You can't fit all the pixels, of course. But if you write code that *tries* to access pixels beyond the edge of the target picture, you will get an error about `OutOfBoundsException`. This statement prevents that.

The conditional says that we *only* get the target pixel and set its color, if the *x* and *y* values that we're going to access are less than the maximum width of the target and the the maximum height of the target. We stay well inside the boundary of the picture that way<sup>1</sup>.

So far, we've only only seen methods that return `void`. We get some amazing expressive power by combining methods that return other objects. Below is an example of how we use the methods in class `Picture` to scale a picture larger (or smaller).

```
> // Make a picture from a file selected by the user
> Picture flower = new Picture(FileChooser.getMediaPath("flower1.jpg"));
> Picture bigFlower = flower.scale(2.0);
> bigFlower.show();
> bigFlower.write("bigFlower.jpg"); // write out the changed picture
```

<sup>1</sup>Of course, if you try to compose to the *left* of the picture, or *above* it, by using negative starting index values, you will get an exception still.

Program  
Example #16

Example Java Code: **Method to scale the Picture by a factor**

```

2      /**
3       * Method to scale the picture by a factor, and return the result
4       * @param factor to scale by (1.0 stays the same,
5       *    0.5 decreases each side by 0.5, 2.0 doubles each side)
6       * @return the scaled picture
7       */
8      public Picture scale(double factor) {
9
10         Pixel sourcePixel, targetPixel;
11         Picture canvas = new Picture(
12             (int) (factor*this.getWidth()+1),
13             (int) (factor*this.getHeight()+1));
14         // loop through the columns
15         for (double sourceX = 0, targetX=0;
16             sourceX < this.getWidth();
17             sourceX+=(1/factor), targetX++) {
18
19             // loop through the rows
20             for (double sourceY=0, targetY=0;
21                 sourceY < this.getHeight();
22                 sourceY+=(1/factor), targetY++) {
23
24                 sourcePixel = this.getPixel((int) sourceX,(int) sourceY);
25                 targetPixel = canvas.getPixel((int) targetX, (int) targetY);
26                 targetPixel.setColor(sourcePixel.getColor());
27             }
28         }
29     }
30     return canvas;
31 }

```

**How it works:** The method `scale` takes as input the amount to scale the picture `this`. This method is declared type `Picture`, instead of `void`—`scale` returns a picture.

The basic process of scaling isn't too complicated. If we have a picture and want it to fit into a smaller space, we have to lose some pixels—we simply can't fit all the pixels in. (All pixels are basically the same size, for our purposes.) One way of doing that is to skip, say, every other pixel, by skipping every other row and column. We do that by adding two to each index instead of incrementing by one each time through the loop. That reduces the size of the picture by 50% in each dimension.

What if we want to scale *up* a picture to fill a large space? Well, we have to duplicate some pixels. Think about what happens if we add 0.5 to the index variable (either for  $x$  or  $y$ ) each time through the loop. The values that the index variable will take will be 0, 0.5, 1.0, 1.5, 2.0, 2.5, and so on. But the index variable can only be an integer, so we'd chop off the decimal. The result is 0, 0, 1, 1, 2, 2, and so on. By adding 0.5 to the index variable, we end up taking each position twice, thus doubling the size of the picture.

Now, what if we want a different sizing—increase by 30% or decrease by 25%? That's where the factor comes in as the input to scale. If you want a factor of 0.25, you want the new picture to be 1/4 of the original picture in each dimension. So what do you add to the index variable? It turns out that  $1/\text{factor}$  works most of the time. For example,  $1/0.25$  is 4, which is a good index increment to get 0.25 of the size. We said that it works most of the time. Why doesn't it work all of the time? What happens if you try 0.3 as the factor? Does this work?

The scale method starts out by *creating* a target picture. The picture is sized to be the scaling factor times the height and width—so the target will be bigger if the scaling factor is over 1.0, and smaller if it is less. As we can see here, new instances of the class `Picture` can be created by filename *or* by specifying the height and width of the picture. The returned picture is blank. We add one to deal with off-by-one errors on oddly sized pictures.

The tricky part of this method is the **for** loops.

```
for (double sourceX = 0, targetX=0;
      sourceX < this.getWidth();
      sourceX+=(1/factor), targetX++)
```

Like in `compose`, we're manipulating two variables at once in this **for** loop. We're using **double** variables to store the indices, so that we can add a  $1/\text{factor}$  to them and have them work, even if  $1/\text{factor}$  isn't an integer. Again, we start out at zero, and keep going as long as there are columns (or rows, for the  $y$  variable) to process. The increment part has us adding one to `targetX` but doing `sourceX += (1/factor)` for the `sourceX` variable. This is a shortcut that is equivalent to `sourceX = sourceX + (1/factor)`.

When we use the index variables, we cast them to integers, which removes the floating point part.

```
sourcePixel = this.getPixel((int) sourceX, (int) sourceY);
```

At the very of end of this method, we **return** the newly created picture. The power of returning a new picture is that we can now do a lot of manipulation of pictures with opening up only a few pictures and *never changing those original pictures*. Consider the below which creates a mini-collage by creating a new blank picture (by asking for a **new** `Picture` with a height and width as inputs to the constructor, instead of a filename) then composing pictures onto it, scaled at various amounts (Figure 3.4).

```
> Picture blank = new Picture(600,600);
> Picture swan = new Picture(FileChooser.getMediaPath("swan.jpg"));
```

```

> Picture rose = new Picture(FileChooser.getMediaPath("rose.jpg"));
> rose.scale(0.5).compose(blank, 10, 10);
> rose.scale(0.75).compose(blank, 300, 300);
> swan.scale(1.25).compose(blank, 0, 400);
> blank.show();

```

What's going on here? How can we *cascade* methods like this? It's because *all* pictures understand the same methods, whether they were created from a file or created from nothing. So, the *scaled* rose understands *compose* just as well as the rose itself does.

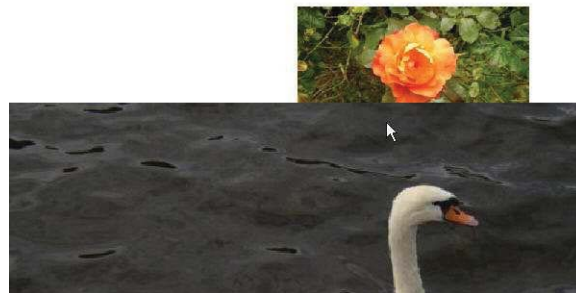
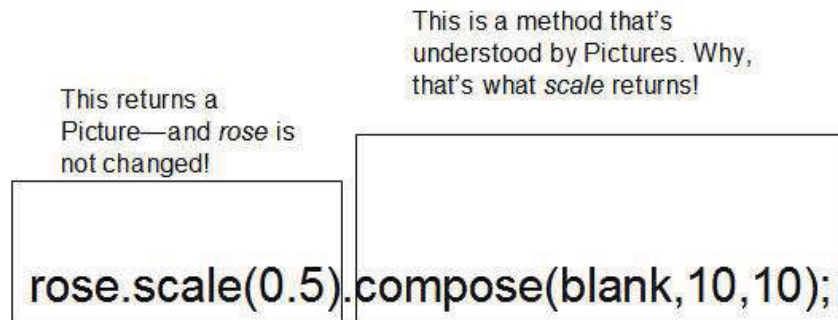


Figure 3.4: Mini-collage created with *scale* and *compose*

Sometimes you don't want to show the result. You may prefer to explore it, which allows you to check colors and get exact *x* and *y* coordinates for

parts of the picture. We can explore pictures to figure out their sizes and where we want to compose them (Figure 3.5).

```
> Picture guy = new Picture(FileChooser.getMediaPath("mRight.jpg"))
> Picture jungle=new Picture(FileChooser.getMediaPath("jungle.jpg"))
> guy.explore()
> jungle.explore()
```

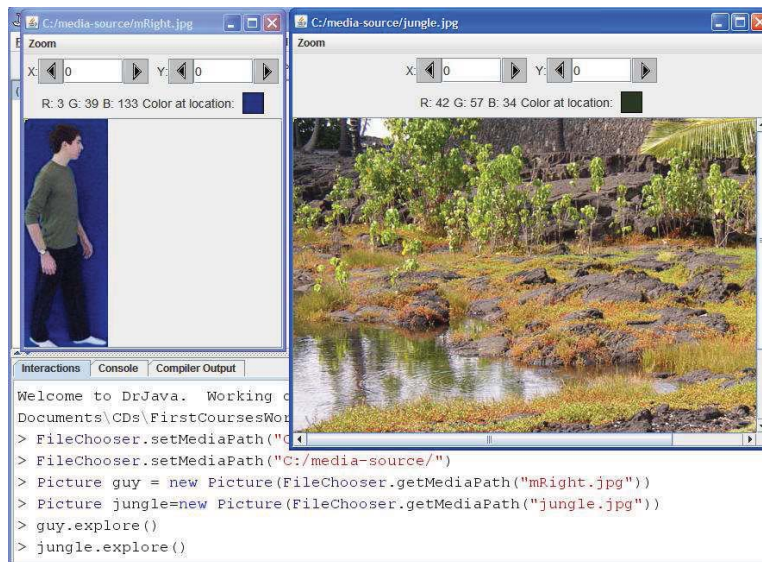


Figure 3.5: Using the explore method to see the sizes of the guy and the jungle

### Composing by Chromakey

*Chromakey* is the video technique by which a meteorologist on our television screen gestures to show a storm coming in from the East, and we see the meteorologist in front of a map (perhaps moving) where the storm is clearly visible in the East next to the meteorologist's hand. The reality is that the meteorologist is standing in front of a blue or green screen. The chromakey algorithm replaces all the blue or green pixels in the picture with pixels of a different background, effectively changing where it looks like the meteorologist is standing. Since the background pixels won't be all the *exact* same blue or green (due to lighting and other factors), we usually use a *threshold* value. If the blue or green is "close enough" (that is, within a threshold distance from our comparison blue or green color), we swap the background color.

There are a couple of different chromakey methods in `Picture`. The method `chromakey()` lets you input the color for the background and a threshold for how close you want the color to be. The method `blueScreen()` assumes that the background is blue, and looks for more blue than red or green (Figure 3.6). If there's a lot of blue in the character (e.g., the guy's denim slacks in the picture), it's hard to get a threshold to work right. It's the same reason that the meteorologists don't wear blue or green clothes—we'd see right through them!

```
> Picture p = new Picture(FileChooser.getMediaPath("mRight.jpg"))
> Picture back = new Picture(640,480)
> p.bluescreen(back,65,250)
> import java.awt.Color // to allows us to use the short name for color
> p.chromakey(back,Color.BLUE,100,165,250)
> p.chromakey(back,Color.BLUE,200,265,250)
> back.show()
```

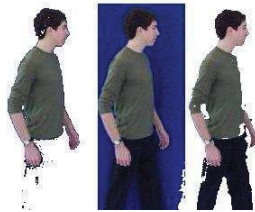


Figure 3.6: Chromakeying the guy onto a blank screen using `blueScreen` and different thresholds for `chromakey`

*Program*  
*Example #17*

**Example Java Code: Methods for general chromakey and blueScreen**

```
/**
2  * Method to do chromakey using an input color for the background
  * and a point for the upper left corner of where to copy
4  * @param target the picture onto which we chromakey this picture
  * @param bgColor the color to make transparent
```

```

6      * @param threshold within this distance from bgColor, make transparent
7      * @param targetX target X position to start at
8      * @param targetY target Y position to start at
9      */
10     public void chromakey(Picture target, Color bgColor, int threshold,
11                           int targetX, int targetY) {
12
13         Pixel currPixel = null;
14         Pixel newPixel = null;
15
16         // loop through the columns
17         for (int srcX=0, trgX=targetX;
18             srcX<getWidth() && trgX<target.getWidth();
19             srcX++, trgX++) {
20
21             // loop through the rows
22             for (int srcY=0, trgY=targetY;
23                 srcY<getHeight() && trgY<target.getHeight();
24                 srcY++, trgY++) {
25
26                 // get the current pixel
27                 currPixel = this.getPixel(srcX,srcY);
28
29                 /* if the color at the current pixel is within threshold of
30                  * the input color, then don't copy the pixel
31                  */
32                 if (currPixel.colorDistance(bgColor)>threshold) {
33                     target.getPixel(trgX, trgY).setColor(currPixel.getColor());
34                 }
35             }
36         }
37     }
38
39     /**
40     * Method to do chromakey assuming a blue background
41     * @param target the picture onto which we chromakey this picture
42     * @param targetX target X position to start at
43     * @param targetY target Y position to start at
44     */
45     public void blueScreen(Picture target,
46                           int targetX, int targetY) {
47
48         Pixel currPixel = null;
49         Pixel newPixel = null;
50
51         // loop through the columns
52         for (int srcX=0, trgX=targetX;
53             srcX<getWidth() && trgX<target.getWidth();
54             srcX++, trgX++) {

```

```

56     // loop through the rows
57     for (int srcY=0, trgY=targetY;
58         srcY<getHeight() && trgY<target.getHeight();
59         srcY++, trgY++) {
60
61         // get the current pixel
62         currPixel = this.getPixel(srcX,srcY);
63
64         /* if the color at the current pixel mostly blue (blue value is
65          * greater than red and green combined), then don't copy pixel
66          */
67         if (currPixel.getRed() + currPixel.getGreen() >
68             currPixel.getBlue()) {
69             target.getPixel(trgX, trgY).setColor(currPixel.getColor());
70         }
71     }
72 }

```

### 3.4 Creating classes that do something

So far, we have created methods in the class `Picture` that know *how* to do something, but we actually *do* things with statements in the Interactions Pane. How do we get a Java class to *do* something? We use a particular method that declares itself to be the main thing that this class *does*. You declare a method like this:

```

public static void main(String[] args) {
    //code goes here
}

```

The code that goes inside a main method is exactly what goes in the Interactions Pane. For example, here's a class that the *only* thing it does is to create a mini-collage.

*Program*  
*Example #18*

Example Java Code: **A public static void main in a class**

```

public class MyPicture {
2
3     public static void main(String args[]){
4
5         Picture canvas = new Picture(600,600);
6         Picture swan =
7             new Picture(FileChooser.getMediaPath("swan.jpg"));
8         Picture rose =
9             new Picture(FileChooser.getMediaPath("rose.jpg"));

```

```

10     Picture turtle =
        new Picture(FileChooser.getMediaPath("turtle.jpg"));
12
14     swan.scale(0.5).compose(canvas,10,10);
        swan.scale(0.5).compose(canvas,350,350);
        swan.flip().scale(0.5).compose(canvas,10,350);
16     swan.flip().scale(0.5).compose(canvas,350,10);
        rose.scale(0.25).compose(canvas,200,200);
18     turtle.scale(2.0).compose(canvas,10,200);
        canvas.show();
20 }
    }

```

The seemingly-magical incantation **public static void** main(String [] args) will be explained more later, but we can talk about it briefly now.

- **public** means that it's a method that any other class can access.
- **static** means that this is a method accessible from the *class*. We don't need to create instances (objects) of this class in order to run the main method.
- **void** means that the main method doesn't return anything.
- String[] args means that the main method can actually take inputs *from the command line*. You can run a main method from the command line by typing the command java and the class name, e.g. java MyPicture (presuming that you have Java installed!).

To run a main method from within DrJava, use function key F2. That's the same as using RUN DOCUMENT'S MAIN METHOD from the TOOLS menu (Figure 3.7). Or just click on the RUN button.



Figure 3.7: Run the main method from DrJava

A main method is not very object-oriented – it's not about defining what an object knows or what it can do. But it is pretty useful.

## Exercises

1. What index would you use to get the 2nd element from an array?  
What index would you use to get the 10th element from an array?

2. What is output by the following code?

```
int[] myArray = {3, 8, 2, 7, 5};
System.out.println(myArray[3]);
System.out.println(myArray[5]);
```

3. What is the result of  $1 / 3$ ? Can you explain why you get this result?
4. What is the result of  $4 \% 2$ ? What is the result of  $5 \% 2$ ? What does  $\%$  do?
5. What is output from the following code?

```
for (int x = 0; x < 5; x++) {
    System.out.println(x);
}
```

6. What is output from the following code?

```
int x = 0;
while (x < 10) {
    System.out.println(x);
    x++;
}
```

7. Which loop would you use if you just want to loop through all the pixels in a picture and change each one? Which loop would you use if you want to read all the lines from a file until the method reading lines returns null? While loop would you use if you want to sum all the numbers from 1 to 100?
8. What class does the class `Sound` inherit from? What class does the class `SimplePicture` inherit from?
9. Look at the documentation in the `doc` folder of the `java-source` directory. Find the documentation for the `Picture` class. How many constructors does it have?
10. Look at the documentation in the `doc` folder of the `java-source` directory. Find the documentation for the `Sound` class. How many constructors does it have?
11. How many constructors can you find in `Picture.java`? How many constructors can you find in `Sound.java`?
12. Write a method that creates a black and white picture from a color picture. You can simply set the red, green, and blue values in the picture to the average of the original red, green, and blue values.
13. Write a method that creates the negative of the current picture. You can do this by setting the red value to  $255 - \text{currPixel.getRed}()$ . Do the same for the green and blue.

14. Write a method that flips the pixels in the picture upside down, or 180 degrees.
15. Write a method that creates and returns a new picture with the pixels rotated 90 degrees to the left.
16. Write a method that creates and returns a new picture with the pixels rotated 90 degrees to the right.
17. Write a method that rotates the pixels in a picture 90 degrees to the left.
18. Write a method to decrease the red in only the top half of the picture.
19. Write a method to increase the red that takes as input a number to use as the amount to multiply the current red value by.
20. Write a method to scale only the top half of the picture.
21. What happens when you call scale using a factor like a third? Does it still work correctly?
22. Write a method that copies part of the source picture onto the target picture. Specify the region to copy using the x and y values for the top left and bottom right of the region to copy.
23. Write a method that does chromakey on only a part of the source picture. Specify the region to do chromakey on using the x and y values for the top left and bottom right of the region to copy.



## 4 Objects as Agents: Manipulating Turtles

### Chapter Learning Objectives

We are going to model our wildebeests and villagers as *agents*—objects that behave independent of each other, seemingly simultaneously, with a graphical (visible) representation. Turtles are an old computational idea that are useful for understanding agents behavior. They are also a powerful tool for understanding object-oriented programming. In this chapter, we learn about turtles in order to simplify our later animations and simulations.

#### The computer science goals for this chapter are:

- To introduce some of the history of object-oriented programming, from Logo (and turtles) to Smalltalk.
- To generalize an understanding of objects, from Pictures to Turtles.
- To better understand cascading methods.
- To introduce some basic list manipulation ideas, e.g., that nodes are different *objects*.
- To use *exceptions* in order to use sleep.

#### The media learning goals for this chapter are:

- To create animations using a FrameSequencer.
- To add another technique for composing pictures.
- To use a simple technique for rotating pictures.

### 4.1 Turtles: An Early Computational Object

In the mid-1960's, Seymour Papert at MIT and Wally Feurzeig and Danny Bobrow at BBN Labs were exploring educational programming by children. That was a radical idea at the time. Computers were large, expensive devices which were shared by multiple people at once. Some found

the thought of giving up precious computing time for use by 10 or 11 year old children to be ludicrous. But Papert, Feurzeig, and Bobrow believed that the activity of programming was a great context for learning all kinds of things, including learning about higher-order thinking skills, like planning and debugging. They created the programming language *Logo* as a simplified tool for children.

The most common interaction with computers in those days was through teletypes—large machines with big clunky keys that printed all output to a roll of paper, like a big cash register receipt paper roll. That worked reasonably well for textual interactions, and much of the early use of Logo was for playing with language (e.g., writing a pig-Latin generator). But the Logo team realized that they really needed some graphical interaction to attract the kids with whom they were working. They created a robot *turtle* with an attached pen to give the students something to control to make drawings.

The simple robot turtle sat on a large piece of paper, and when it moved (and if its pen was “down” and touching the paper) it would draw a line behind it. The Logo team literally invented a new kind of mathematics to make Logo work, where the turtle didn’t know Cartesian coordinates ( $(x, y)$  points) but instead knew its heading (which direction it was facing), and could turn and go forward. This relative positioning (as opposed to global, Cartesian coordinates) was actually enough to do quite a bit of mathematics, including biological simulations and an exploration of Einstein’s Special Theory of Relativity[1].

As we will see in the next section, the Logo turtle is very clearly a computational object. The turtle *knows* some things (like its heading and whether its pen is down) and it can *do* some things (like turn and go forward). But even more directly, the Logo turtle influenced the creation of object-oriented programming. Alan Kay[4] modeled his *Smalltalk* programming language on Logo—and Smalltalk is considered to be the very first object-oriented programming language, and Alan Kay is considered to be the inventor of object-oriented programming.

The Logo turtle still exists in many implementations in many languages. Seymour Papert’s student, Mitchel Resnick, developed a version of Logo, *StarLogo* with thousands of turtles that can interact with one another. Through this interaction, they can simulate scenarios like ants in an anthill, or termites, slime mold, or vehicle traffic[5].

## 4.2 Drawing with Turtles

We’re going to use turtles to draw on our pictures in interesting and flexible ways, and to simplify animation. Our Turtle class instances (objects) can be created on a Picture or on a World. Think of a World as a constantly updating picture that repaints automatically. We create a World by simply creating a **new** one. We create a Turtle on this world by passing the World

object in as input to the Turtle constructor (Figure 4.1).

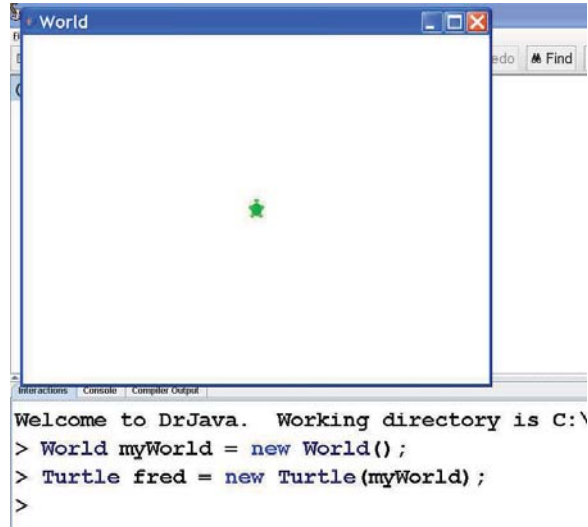


Figure 4.1: Starting a Turtle in a new World

Here’s an example of opening a turtle on a Picture instead (Figure 4.2). Turtles can be created on blank Picture instances (which start out white) in the middle of the picture with pen down and with black ink. When a turtle is told to go forward, it moves forward the input number of *turtle steps* (think “pixels,” which isn’t *exactly* correct, but is close enough most of the time—the actual unit is computed by Java depending on your screen resolution) in whatever direction the turtle is currently facing. You can change the direction in which the turtle is facing by using the turn method which takes as input the number of degrees to turn. Positive degrees are clockwise, and negative ones are counter-clockwise.

```
> Picture blank = new Picture(200,200);
> Turtle fred = new Turtle(blank);
> fred
No name turtle at 100, 100 heading 0.0.
> fred.turn(-45);
> fred.forward(100);
> fred.turn(90);
> fred.forward(200);
> blank.show();
> blank.write("c:/Temp/turtleExample.jpg");
```

Turtles know their position (unlike the original robot turtles) and their heading. The heading is 0 when they point north (how they’re first created), and 90 when pointed to the right (due east), and 180 when pointed

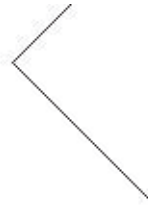


Figure 4.2: A drawing made by a turtle

straight down (south). Clearly, turtles know things (e.g., their heading,  $x$  and  $y$  position) and can do things (e.g., like move forward and turn).

```
> fred.forward(100);
> fred.turn(90);
> fred.getHeading()
90
> fred.getXPos()
320
> fred.getYPos()
140
```

Turtles can pick up their pen (stop drawing) using either the method `penUp()` or the code `setPenDown(false)`. We can set the pen down using `penDown()` or `setPenDown(true)`. Java does know about truth, or at least, about *Boolean* values: **true** and **false**.

To draw more complex shapes, we tell the turtle to do its basic steps repeatedly. Telling a turtle to go forward a certain number of steps and to turn 90 degrees 4 times draws a square.

```
> for (int sides=0; sides < 4 ; sides++) {
    fred.forward(100); fred.turn(90);
}
```

### When cascades don't work

Here's a thought experiment: will this work?

```
> World earth = new World();
> Turtle turtle = new Turtle(earth);
> turtle.forward(100).right(90);
```

The answer is “no,” but can you figure out why? Here’s a hint: The error you get in the Interactions Pane is shown below.

```
Error: No
'right' method in 'void' with arguments: (int)
```

The error message is actually telling you exactly what the problem is, it’s written in *Javanese*—it presumes that you understand Java and can thus interpret the message.

The problem is that the forward method does not return anything, and certainly not a *turtle*. The method forward returns **void**. When we cascade methods like this, we are telling Java to invoke right(90) on what turtle.forward(100) returns. Since forward returns **void**, Java checks if instances of class **void** understand right. Of course not—**void** is nothing at all. So Java tells us that it checked for us, and the class **void** has no method right that takes an integer (**int**) input (e.g., 90 in our example). (Of course, **void** doesn’t know anything about right with *any* inputs, but just in case we only got the inputs wrong, Java lets us know what it looked for.) Thus: Error: No ‘right’ method in ‘void’ with arguments: (**int**).

You can only use a cascade of method calls if the *previous* method call returns an object that has a method defined for the *next* method call. Since forward returns nothing (**void**), you can’t cascade anything after it. Sure, we could create forward so that it does return the turtle **this**, the one it was invoked on, but does this make any sense? Should forward return something?

## Making lots of turtles

Using Mitchel Resnick’s StarLogo as inspiration, we may want to create something with *lots* of turtles. For example, consider what this program draws on the screen.

Example Java Code: **Creating a hundred turtles**

*Program  
Example #19*

```
public class LotsOfTurtles {
2
    public static void main(String[] args) {
4        // Create a world
        World myWorld = new World();
6        // A flotilla of turtles in an array
        Turtle [] myTurtles = new Turtle[100];
8
        // Make a hundred turtles
10       for (int i=0; i < 100; i++) {
            myTurtles[i] = new Turtle(myWorld);
12    }
}
```

```

14 //Tell them all what to do
   for (int i=0; i < 100; i++) {
16 // Turn a random amount between 0 and 360
   myTurtles[i].turn((int) (360 * Math.random()));
18 // Go 100 pixels
   myTurtles[i].forward(100);
20 }
   }
22 }

```

**How it works:** Study the program and think about it before you look at Figure 4.3.

- First we create a World and name it myWorld.
- Next, we create an array to store 100 Turtle instances. Notice that `Turtle [] myTurtles = new Turtle[100];` creates no turtles!. That 100 is enclosed in square brackets—we’re not calling the Turtle constructor yet. Instead, we’re simply asking for 100 slots in an array myTurtles that will each refer to a Turtle.
- Inside a **for** loop that goes 100 times, we create each turtle and put a reference to it in the array at the current index using the code: `myTurtles[i] = new Turtle(myWorld);`.
- Finally, we tell each of the turtles to turn a random amount and go forward 100 steps. `Math.random()` returns a number between 0 and 1.0 where all numbers (e.g., 0.2341534) in that range are equally likely. Since that will be a **double**, we have to cast the result to **int** to use it as an input to forward.

Figured it out yet? It makes a circle of radius 100! This is an example from Mitchel Resnick’s book that introduced StarLogo[5].

Obviously, we can have more than one Turtle in a World at once. Instances of class Turtle know some methods that allow them to interact with one another. They know how to `turnToFace(anotherTurtle)` which changes one turtles heading to point to the other turtle. They also know how to compute `getDistance(x,y)` which is the distance from this turtle (the one that `getDistance` was invoked upon) to the point  $x, y$ . Thus, in this example, `r2d2` goes off someplace random on `tattoine`, but `c3po` turns to face him and moves forward exactly the right distance to catch `r2d2`.

```

> World tattoine = new World();
> Turtle r2d2 = new Turtle(tattoine);
> r2d2.turn((int)
   (360 * Math.random()));

```

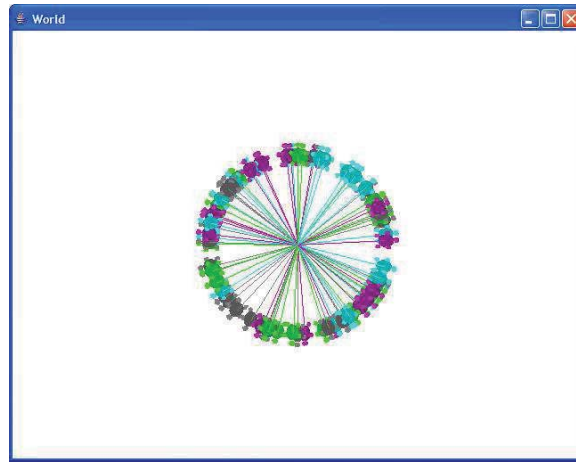


Figure 4.3: What you get with a hundred turtles starting from the same point, each turning a random amount from 0 to 360, then moving forward the same amount

```
> r2d2.forward((int)
    (Math.random() * 100));
> Turtle c3po = new Turtle(tattoine);
> c3po.turnToFace(r2d2);
> c3po.forward((int)
    (c3po.getDistance(
        r2d2.getXPos(), r2d2.getYPos())));
```

### Composing pictures with turtles

We saw earlier that we can place turtles on an instance (object) of class `Picture`, not just instances (objects) of class `World`. We can also use turtles to compose pictures into other pictures, through use of the `drop` method. Pictures get “dropped” *with the upper left corner of the picture at the center of the turtle and rotated to match the turtle’s heading*. If the turtle is facing down (heading of 180.0), then the picture will be upside down. (Figure 4.4).

```
> Picture monster =
    new Picture(FileChooser.getMediaPath("mscary.jpg"));
> Picture newBack = new Picture(400,400);
> Turtle myTurtle = new Turtle(newBack);
> myTurtle.drop(monster);
> newBack.show();
```

We’ll rotate the turtle and drop the picture again (Figure 4.5).



Figure 4.4: Dropping the monster character

```
> myTurtle.turn(180);
> myTurtle.drop(monster);
> newBack.repaint();
```



Figure 4.5: Dropping the monster character after a turtle rotation

We can drop using loops and patterns, too (Figure 4.6). Why don't we see 12 monsters here? Maybe some are blocking the others?

```
> Picture frame = new Picture(600,600);
> Turtle mabel = new Turtle(frame);
> for (int i = 0; i < 12; i++) {
    mabel.drop(monster); mabel.turn(30);
}
```



Figure 4.6: An iterated turtle drop of a monster

We can combine these in a main method to create a more complex image (Figure 4.7).

Example Java Code: **Making a picture with dropped pictures**

*Program  
Example #20*

```

public class MyTurtlePicture {
2
  public static void main(String [] args) {
4    Picture canvas = new Picture(600,600);
    Turtle jenny = new Turtle(canvas);
6    Picture lilTurtle =
      new Picture(
8      FileChooser.getMediaPath("Turtle.jpg"));
10   for (int i=0; i <=40; i++) {
12     if (i < 20) {
        jenny.turn(20);
14     }
      else {
16     jenny.turn(-20);
        }
18     jenny.forward(40);
20     jenny.drop(lilTurtle.scale(0.5));
    }
22   canvas.show();
  }
24 }

```



Figure 4.7: Making a more complex pattern of dropped pictures

### 4.3 Creating animations with turtles and frames

Our eyes tend to present an image to our brain, even for a few moments after the image as disappeared from sight. That’s one of the reasons why we don’t panic when we naturally blink (many times a minute without noticing)—the world doesn’t go away and we don’t see blackness. Rather, our eyes persist in showing the image in the brief interval when we blink—we call that *persistence of vision*.

A movie is a series of images, one shown right after the other. If we can show at least 16 images (*frames*) in a logical sequence in a second, our eye merges them through persistence of vision, and we perceive continuous motion. Fewer frames per second may be viewed as continuous, but it will probably be choppy. If we show frames that are not in a logical sequence, we perceive a montage, not continuous motion. Typical theater movies present at 24 frames per second, and video is typically 30 frames per second.

If we want to create an animation, then, we need to store a bunch of instances of `Picture` as frames, then play them back faster than 16 frames

per second. We have a class for doing this, it is called `FrameSequencer`.

- The constructor for `FrameSequencer` takes a path to a directory where frames will be stored as JPEG images, so that you can reassemble them into a movie using some other tool (e.g., Windows Movie Maker, Apple Quicktime Player, ImageMagick). Or you can write a Quicktime movie or AVI movie from a directory of frames using the class `MoviePlayer` which is provided in `java-source`.
- A `FrameSequencer` knows how to `show()`. Once shown, a `FrameSequencer` will show each frame as it is added to the `FrameSequencer`. When you first tell a `FrameSequencer` to show, it will warn you that there's nothing to see until a frame is added.
- It knows how to `addFrame(aPicture)` to add another frame to the `FrameSequencer`.
- It knows how to `play(framesPerSecond)` to show a sequence of frames back again. The `framesPerSecond` is the number of frames to show per second. If you specify 16 frames per second or higher this will be perceived as continuous motion.

Here's a silly example of how you might use a `FrameSequencer`. We're adding three (unrelated) pictures to a `FrameSequencer` via `addFrame`. We can then play the frame sequence, one frame per second (or strictly, one frame per 1000 milliseconds).

```
> FrameSequencer f = new FrameSequencer("c:/Temp");
> f.show()
There are no frames to show yet. When you add a frame it will be
shown
> Picture t = new
    Picture(FileChooser.getMediaPath("turtle.jpg"));
> f.addFrame(t);
> Picture barb = new
    Picture(FileChooser.getMediaPath("barbara.jpg"));
> f.addFrame(barb);
> Picture katie = new
    Picture(FileChooser.getMediaPath("katie.jpg"));
> f.addFrame(katie);
> f.play(1); // display one frame per second
```

Let's combine turtles and a `FrameSequencer` to make an animation of frames.

Example Java Code: **An animation generated by a Turtle**

*Program  
Example #21*

```

public class MyTurtleAnimation {
2
    private Picture canvas;
4    private Turtle jenny;
    private FrameSequencer f;
6
    public MyTurtleAnimation() {
8
        canvas = new Picture(600,600);
10       jenny = new Turtle(canvas);
        f = new FrameSequencer("C:/Temp/");
12    }

14    public void next() {
        Picture lilTurtle =
16        new Picture(FileChooser.getMediaPath("Turtle.jpg"));
        jenny.turn(-20);
18        jenny.forward(30);
        jenny.turn(30);
20        jenny.forward(-5);
        jenny.drop(lilTurtle.scale(0.5));
22        f.addFrame(canvas.copy()); // Try this as
                                    // f.addFrame(canvas);
24    }

26    public void next(int numTimes) {
        for (int i=0; i < numTimes; i++) {
28            this.next();
        }
30    }

32    public void show() {
        f.show();
34    }

36    public void play(int framesPerSecond) {
        f.show();
38        f.play(framesPerSecond);
    }
40 }

```

We run this program like this:

```

> MyTurtleAnimation anim = new MyTurtleAnimation();
> anim.next(20); // Generate 20 frames
> anim.play(2); // Play them back, two per second

```

**How it works:** An instance of `MyTurtleAnimation` has three instance variables associated with it: A `Picture` onto which the turtle will draw named

canvas, a Turtle named jenny, and a FrameSequencer. The constructor for MyTurtleAnimation creates the original objects for each of these three names.

**Common Bug: Don't declare the instance variables (fields) in methods or constructors**

There's a real temptation to put "Picture" in front of that line in the constructor `canvas = new Picture(600,600);`. But resist it. Will it compile? Absolutely. Will it work? Not at all. If you declare `canvas` as a `Picture` inside of the constructor `MyTurtleAnimation()`, it *only* exists in the constructor. You want to access the field `canvas` which exists outside that method. If you don't declare it in the constructor, Java figures out that you mean the field (instance variable). It assumes that you mean `this.canvas` which is the field `canvas` in the current object.

There are two different `next` methods in `MyTurtleAnimation`. The one that we called in the example (where we told it to go for 20 steps) is this one:

```
public void next(int numTimes) {
    for (int i=0; i < numTimes; i++) {
        this.next();
    }
}
```

All that `next(int numTimes)` does is to call `next()` the input `numTimes` number of times. So the real activity in `MyTurtleAnimation` occurs in `next()` with no inputs.

```
public void next() {
    Picture lilTurtle =
        new Picture(FileChooser.getMediaPath("Turtle.jpg"));
    jenny.turn(-20);
    jenny.forward(30);
    jenny.turn(30);
    jenny.forward(-5);
    jenny.drop(lilTurtle.scale(0.5));
    f.addFrame(canvas.copy()); // Try this as
                               // f.addFrame(canvas);
}
```

The method `next()` does a bit of movement, a drop of a picture, and an addition of a frame to the `FrameSequencer`. Basically, it generates the next frame in the animation sequence.

The `show()` and `play()` methods *delegate* their definition to the `FrameSequencer` instance. Delegation is where one class accomplishes what it needs to do by asking an instance of another class to do the task. It's perfectly reasonable to ask an animation to `show()` or `codeplay()`, but the way that it would

the animation accomplishes those tasks is by asking the FrameSequencer to show or play.

### The Data structure within FrameSequencer

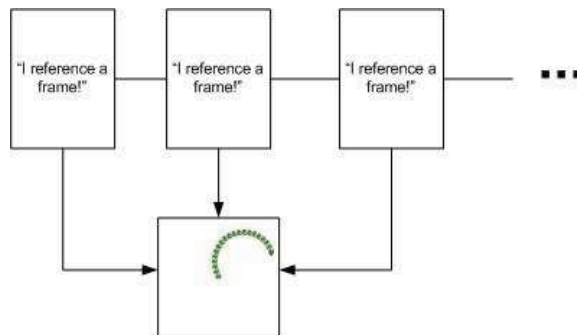
Did you try this same animation with the last line of next changed to `f.addFrame(canvas)`? What happened? If you did try it, you may have thought you made a mistake. When you ran the next animation, you saw the animation play out as normal. But when you executed play, you saw only the final frame appear—never any other frame. Go check the temporary directory where the FrameSequencer wrote out the frames. You'll find a bunch of JPEG images there: `frame0001.jpg`, `frame0002.jpg`, and so on. So the animation did work and the FrameSequencer did write out the frames. But why isn't it replaying correctly?

What we are seeing helps us to understand how FrameSequencer works and what its internal data structure is. As you might imagine, a FrameSequencer is a series of frames—but that's not correct in the details. The FrameSequencer is actually a list of *references* to Picture objects. Each element in the FrameSequencer refers to some Picture.



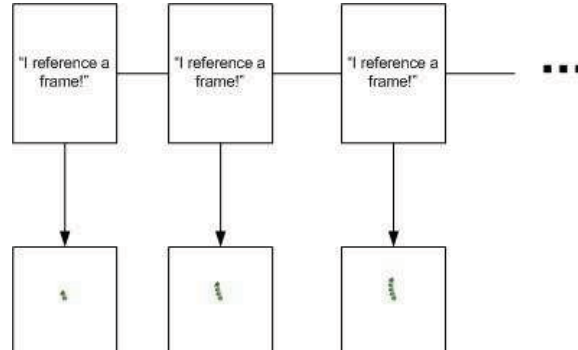
If a FrameSequencer does not actually have any frames in it, where do the frames come from? From the Picture that you input to `addFrame`! That's what the FrameSequencer frame references point to.

Without the `.copy()` method call on `canvas`, all the references in the code-FrameSequencer point to *one* Picture, `canvas`. There is only *one* picture in the FrameSequencer, and since, at the end, the `canvas` is in its final state, then all the references in FrameSequence point to that same `canvas` Picture in its final state.



With `canvas.copy()`, you create a copy of the picture in the `canvas`, a *different* Picture for *each* reference in the FrameSequencer. When we tell the

FrameSequencer to play, the pictures referenced by the FrameSequencer play out on the screen.



#### 4.4 Making a Slow Moving Turtle with sleep and exceptions

Let's say that you wanted a turtle that would move *slowly*, that would pause after moving so that you could more easily watch it move. One way to do that would be to use `blockingPlay` on a sound to pause execution. Fortunately, there's an easier way.

The flow of execution in Java is called a *thread*. The currently executing thread is called `Thread` (capital letter). We can tell the `Thread` to sleep, or to pause for a certain number of milliseconds. Executing `Thread.sleep(1000)` would pause execution for one second.

However, we can't just *call* `Thread.sleep`. Java wants you to write code that takes care of bad things that might happen in your code. If you use `Thread.sleep`, you *have* to take care of *exceptions* or else Java won't even compile your code.

Exceptions are disruptions in the normal flow of a program. There is actually a Java class named `Exception` that is used in handling exceptions. In other programming languages, there are ways for programmers to check if something bad has happened—and most programmers don't check. Java *requires* the programmer to handle exceptions. The programmer can be specific (e.g., "If the disk fails, do this. If the filename is bad, do that.") or general (e.g., "If *anything* bad happens, here's how to bail out.").

There is a special Java statement just for handling exceptions. It's called **try-catch**. It looks something like this:

```
try {
  \\code that can cause the exceptions
} catch (ExceptionClassName varName) {
  \\code to handle this exception
} catch (ExceptionClassName varName) {
  \\code to handle that exception
}
```

You can deal with (“catch”) several exceptions at once, of different types. If you do try to distinguish between exceptions in a single statement like that, put the most general one last, e.g., catch the general `Exception`, and maybe one like `FileNotFoundException` (for trying to open a file for reading that doesn’t exist) earlier in the list. All those other exceptions are subclasses of `Exception`, so catching `Exception` will handle all others. A general exception handling **try–catch** might look like this:

```

try {
  //code that can throw the exception
} catch (Exception e) {
  System.err.println("Exception: " + e.getMessage());
  System.err.println("Stack Trace is:");
  e.printStackTrace();
}

```

That `e` in the above code is a variable that will be a reference to the exception object, if an exception occurs. Exceptions know several different methods. One returns (as a `String`) the error message associated with the exception: `e.getMessage()`. Another prints out the line where the exception occurred in the currently executing method and all the method calls between the beginning of execution and the exception. `e.printStackTrace()`.

Since `Thread.sleep` can throw exceptions, we have to be use **try–catch** around it.

Program  
Example #22

Example Java Code: **SlowWalkingTurtle**

```

2  /**
3   * A class that represents a slow walking turtle
4   * @author Mark Guzdial
5   * @author Barb Ericson
6   */
7  class SlowWalkingTurtle extends Turtle {
8
9      /** Constructor that takes the model display
10     * @param modelDisplay the thing that displays the model
11     */
12     public SlowWalkingTurtle (ModelDisplay modelDisplay) {
13         // let the parent constructor handle it
14         super(modelDisplay);
15     }
16
17     /**
18     * Method to have the turtle go forward then wait a moment
19     * and then go on.
20     * @param pixels the number of pixels to go forward
21     */

```

```

22     public void forward(int pixels) {
23         super.forward(pixels);
24         try {
25             Thread.sleep(500); // Wait a half sec
26         } catch (Exception e) {
27             System.err.println("Exception: " + e.getMessage());
28             System.err.println("Stack Trace is:");
29             e.printStackTrace();
30         }
31     }
32 }

```

**How it works:** The `SlowWalkingTurtle` is a kind of `Turtle`. We need a constructor that passes the construction off to the superclass. We override `forward` so that, after moving, we pause for a half second. The rest of that method is our `try-catch` to deal with anything bad that might happen.

We use our slow-moving turtle like this:

```

> World earth = new World();
> SlowWalkingTurtle mark = new SlowWalkingTurtle(earth)
> mark.forward(100); mark.forward(100);

```

## Exercises

1. What direction will the Turtle be facing after the following code executes? What will the x and y position be?

```

World earth = new World();
Turtle tina = new Turtle(earth);
tina.turn(-90);

```

2. What direction will the Turtle be facing after the following code executes? What will the x and y position be?

```

World earth = new World();
Turtle tina = new Turtle(earth);
tina.forward(30);

```

3. What direction will the Turtle be facing after the following code executes? What will the x and y position be?

```

World earth = new World();
Turtle tina = new Turtle(earth);
tina.forward(50);
tina.turn(90);

```

4. Add a method to the `Turtle` class to teach all turtles how to draw a square.

5. Add a method to the Turtle class to teach all turtles how to draw an equilateral triangle.
6. Add a method to the Turtle class to teach all turtles how to draw a pentagon.
7. Add a method to the Turtle class to teach all turtles how to draw a hexagon.
8. Add a method to the Turtle class to teach all turtles how to draw a star.
9. Add a method to the Turtle class to teach all turtles how to draw a letter (like T).
10. Look at the documentation for the Turtle class. How many constructors does it have? What class does it inherit from? How can you change the color of the pen? How can you make the line the Turtle draws thicker? How do you move to the Turtle to a particular position (x and y location)?
11. Add a method to the Turtle class that takes the radius of a circle and drops 4 pictures. It should drop one picture at 0 degrees, one at 90 degrees, one at 180 degrees, and one at 270 degrees on the circle. The pictures should be rotated to match the degrees (so the picture dropped at 90 degrees on the circle should be rotated right 90 degrees).
12. Add a method to the Turtle class that takes the radius of a circle and the number of pictures to drop. It should drop the given number of pictures equidistant on the circle.
13. Add a method to the Turtle class that takes the picture to drop and drops 12 copies of the picture with the turtle turning 30 degrees after each drop.
14. Add a method to the Turtle class that moves the Turtle to a passed x and y location and then drops a passed picture.
15. Write a class that creates an image collage using Turtle objects that drop pictures. Each image collage must have the same picture at least 4 times but the image can be rotated, scaled, flipped, and more.
16. Create a movie using the FrameSequencer and a Turtle. In each frame have the Turtle drop a picture and then move forward 5 pixels.
17. Create a movie using the FrameSequencer and a Turtle. In the first frame of the movie have a picture in the top left corner and in each frame move it by 5 pixels in x and y (towards the bottom right corner).

18. Create a movie using the `FrameSequencer` and a `Turtle`. In the first frame of the movie have a picture in the top right corner and in each frame move it by -5 pixels in x and 5 pixels in y (towards the bottom left corner).
19. Create a movie using the `FrameSequencer` and a `Turtle`. For each movie frame create a new blank picture and create a new `Turtle` on the blank picture at a particular location and then drop a picture. Change the location for each frame by 5 pixels in x.
20. Using sampled sounds and turtles, create a musical dance. Play music (sounds at least?) while the turtles move in patterns.
  - Your piece must last at least 10 seconds.
  - You must have at least five turtles. They can drop pictures, they can leave trails (or not), they can spin slowly, whatever you want them to do. Your turtles must move.
  - You must have at least four different sounds.
  - There must be some interweaving between turtle motion and sounds. In other words, you can't move the five turtles a little, then play 10 seconds of music. Hint: Use `blockingPlay()` instead of `play()` to play the sounds. If you use `play()`, the turtle movement and the sounds will go in parallel, which is nice, but you'll have no way to synchronize playing and moving. The method `blockingPlay()` will keep anything else from happening (e.g., turtle movement) during the playing of the sound.

Implement your dance and sounds in a `TurtleDance` class in a main method.

For extra credit, use MIDI (`JMusic`) instead of sampled sounds. Look up the `Play` object in the `JMusic` docs. `Play.midi(score,false)` will play a score in the background (**false** keeps it from quitting Java after playing). `Play.waitcycle(score)` will block (wait) anything else from happening until the score is done playing essentially, letting you block like `blockingPlay()`.