Contents

Outline

Semester schedule 2

0 Welcome to CS220 3
  0.1 Orientation slides 3
  0.2 About this class (Syllabus elements) 5
  0.3 Acknowledgments 9

1 The stack and the heap (although for the moment, mostly just the stack) 10
  1.1 How Java organizes your computer’s memory 10
  1.2 Recursion 18
  1.3 Discussing execution time 22
  1.4 Improving recursion 25
  1.5 Exceptions 30

2 Classes and objects 42
  2.1 Classes, objects and inheritance 42
  2.2 Abstract classes and methods 58
  2.3 More class forms 70

3 Arrays 75
  3.1 Arrays under the hood 75
  3.2 Search and its cost 77
  3.3 Sorting 81
  3.4 Two-dimensional arrays 90

4 Data structures 95
  4.1 Linked lists 96
  4.2 Stacks and queues 104
  4.3 Doubly-linked lists 110

5 Generics 115
  5.1 Generic classes 115
  5.2 Generic methods 121
  5.3 The iterable interface 123
  5.4 The Comparable interface 129
  5.5 Type parameter bounds 133
  5.6 Java Collections classes 141
  5.7 Hash codes 146

6 Interaction 153
  6.1 Files 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 GUIs</td>
<td>166</td>
</tr>
<tr>
<td>7 More data structures</td>
<td>171</td>
</tr>
<tr>
<td>7.1 Trees</td>
<td>171</td>
</tr>
<tr>
<td>7.2 Graphs</td>
<td>177</td>
</tr>
<tr>
<td>A Closing lecture: The road ahead</td>
<td>181</td>
</tr>
<tr>
<td>B Course policies (Syllabus elements)</td>
<td>184</td>
</tr>
<tr>
<td>C Departmental learning outcomes for CS120 and CS220</td>
<td>192</td>
</tr>
<tr>
<td>D Review exercises</td>
<td>195</td>
</tr>
<tr>
<td>E Hints for selected exercises</td>
<td>229</td>
</tr>
</tbody>
</table>
Provisional CS220 semester schedule

See the course homepage for updates and full assignment details. This schedule will evolve as the semester unfolds; see the course homepage or the online course pack for updates. All deadlines are at **8:00am** except where explicitly indicated on the course homepage/Autolab.

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Assignment</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tuesday, September 3</strong></td>
<td>Lecture 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Lectures meet in Centennial 2205</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Course overview, our process, a bit of administration</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, September 4</strong></td>
<td>Lecture 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Java and your computer’s memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Online Quiz 0 by 5pm</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, September 5</strong></td>
<td>Midterm Test 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Covering CS120 prerequisite material</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, September 6</strong></td>
<td>Administrative matters due at noon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Labs meet in 16 Wing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• How we do the things we do</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 1) — Online Quiz 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, September 10</strong></td>
<td>Lecture 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Recursion</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, September 11</strong></td>
<td>Lecture 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A quantum of labor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, September 12</strong></td>
<td>Lecture 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The big O bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, September 13</strong></td>
<td>• Project 0 due</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, September 16</strong></td>
<td>Lab 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Let’s recarpet the lab</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, September 17</strong></td>
<td>Lecture 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sometimes all you need is a little help from your parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, September 18</strong></td>
<td>Lecture 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Exceptions, making them stop</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, September 19</strong></td>
<td>Lecture 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The exceptions hierarchy</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, September 23</strong></td>
<td>Lecture 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A heaping helping (in a helping heap!) of classes and objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, September 24</strong></td>
<td>Midterm Test 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, September 25</strong></td>
<td>Lecture 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Representational objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, September 26</strong></td>
<td>Lecture 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• How things are, and how we declare them to be</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, October 1</strong></td>
<td>Lecture 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Equalities</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, October 2</strong></td>
<td>Lecture 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Things which are not classes despite a strong and compelling resemblance</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, October 3</strong></td>
<td>Lecture 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Anonymous classes, lambda expressions, inner classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, October 7</strong></td>
<td>Lecture 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Design patterns</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, October 8</strong></td>
<td>Lecture 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Arrays under the hood</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, October 9</strong></td>
<td>Lecture 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Passing functionality-bearing objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, October 10</strong></td>
<td>Lecture 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Search and its cost</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, October 11</strong></td>
<td>Supplementary Homework (Set 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, October 14</strong></td>
<td>Midterm Test 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, October 15</strong></td>
<td>Lecture 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Naive sorting</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, October 16</strong></td>
<td>Lab 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Quietly resizing arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, October 17</strong></td>
<td>Lecture 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Merge sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, October 18</strong></td>
<td>Supplementary Homework (Set 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, October 21</strong></td>
<td>Lecture 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Quicksort</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Project 1 due</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, October 22</strong></td>
<td>Lecture 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Multidimensional arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, October 23</strong></td>
<td>Lecture 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• More great uses of arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, October 24</strong></td>
<td>Lecture 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Topic to be announced</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, October 25</strong></td>
<td>Supplementary Homework (Set 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, October 28</strong></td>
<td>Midterm Test 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, October 29</strong></td>
<td>Lecture 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Self-referencing objects, linked lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, October 30</strong></td>
<td>Lecture 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Turning it around</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, October 31</strong></td>
<td>Lecture 23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• More fun with lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, November 1</strong></td>
<td>Project 2 due</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, November 4</strong></td>
<td>Lecture 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Simmer until reduced</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, November 5</strong></td>
<td>Lecture 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Wrappers</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, November 6</strong></td>
<td>Lab 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Our lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, November 7</strong></td>
<td>Lecture 25</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sentinels, other contents</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, November 8</strong></td>
<td>Supplementary Homework (Set 8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, November 11</strong></td>
<td>Lab 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Stacks, queues</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, November 12</strong></td>
<td>Lecture 26</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Doubly-linked lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, November 13</strong></td>
<td>Lab 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, November 14</strong></td>
<td>Lecture 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Generic classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, November 15</strong></td>
<td>Lab 11 followup</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, November 18</strong></td>
<td>Lecture 28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Generic methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, November 19</strong></td>
<td>Midterm Test 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Project 3 due</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, November 20</strong></td>
<td>Lecture 29</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Iterators</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, November 21</strong></td>
<td>Lecture 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Comparison</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, November 25</strong></td>
<td>Lecture 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Type bounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, November 26</strong></td>
<td>Lecture 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Java collections</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, November 27</strong></td>
<td>Class does not meet today</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• UWL follows Friday’s schedule on the Wednesday immediately before Thanksgiv-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Planning</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, December 2</strong></td>
<td>Lecture 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Hash codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Supplementary Homework (Set 11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, December 3</strong></td>
<td>Lecture 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The file and directory model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Preparatory Homework (Set 11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, December 4</strong></td>
<td>Lecture 35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Streams</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thursday, December 5</strong></td>
<td>Lecture 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Binary content</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Monday, December 9</strong></td>
<td>Lab 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Using files</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, December 10</strong></td>
<td>Lecture 37</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Topic to be determined</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wednesday, December 11</strong></td>
<td>Lecture 38</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The road ahead</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deadlines: Project 4 due</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Friday, December 13</strong></td>
<td>Lab 12 followup</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Saturday, December 14</strong></td>
<td>Final examination</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12:15-2:15 p.m. for Sec. 1 (8:50am)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tuesday, December 17</strong></td>
<td>Final examination</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2:30-4:30 p.m. for Sec. 2 (9:55am)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
0 Welcome to CS220

0.1 Orientation slides

The Software Design sequence

• Design algorithms
  – High-level problem-solving skills

• Implement algorithms as programs
  – Java - A modern programming language
  – Organize data and instructions
  – In both algorithms and programs, we must use low-level and precise logic
    * No ambiguity allowed
  – Debugging and testing

• Understand what programs will do

• Communicate technical information about your programs

• Learn how to operate as a technical professional

Software Design I

The six things a program can do

1. Get input
2. Give output
3. Do arithmetic
4. Update a stored value
5. Test a condition, and select an alternative
6. Repeat a group of actions

Ways Java helps you organize your work

• Defining sequences of operations as methods

• Group related data together in arrays and class fields

• Associate data with operations relevant to that data as object methods
Software Design II

• Starting to understand how Java manages your computer’s memory
  – Features of Java built on this understanding: exceptions and recursion

• Similarly, looking under the hood at Java arrays
  – How their place in memory explains the behavior we can observe
  – Multi-dimensional arrays, and how Java really implements them

• A deeper look at inheritance and object-oriented design
  – Abstract vs. concrete, and all of the ways Java allows us to use this distinction
  – When we can and cannot (and should and should not) use inheritance

• Combining classes into data structures

• Applying code to multiple situations via Java generics

• Interacting outside of Java’s world via the file system

Focus on the personal process of creating software

• What does "correct" mean for software?

• How can you tell if your work is correct?

It is not enough to write software — you must build quality in by

• Keeping steps simple

• Keeping your goals clear
  – Recording what your goals are
  – Measuring whether you meet them
  – And both from the beginning of your work

Our process

0. Outline the result

• Set up stubs of classes and methods with the correct signatures
• But empty bodies and dummy results
• "Step 0" means it is very basic, should not be very taxing, and should be an automatic "just do it"

1. Define success, and be ready to measure it

• Set up a main method (in an assigned class, or in a separate one)
• Fill it with uses of the required classes and methods
• You are writing experiments
  – Does your code have the right behavior?
  – Your main should announce desired behaviors, and actual results
    – Every behavior required in the specifications of work which I give you should be examined at least once
• Set up your program to tell you whether it works
2. **Make a plan**
   - Write down how will implement your stub methods
   - Major steps, goals along the way
   - Break down complicated methods by describing different steps as their own methods
   - Write the method steps down as comments in each method’s body
     - A checklist of what you must implement for each method

3. **Implement and debug the methods**
   - Only with testable correctness criteria and a detailed plan in place should you start actually writing the Java implementations of your method bodies

   Do not skip steps!

0.2 **About this class (Syllabus elements)**

0.2.1 **Key facts**

- **Class name**: Software Design II
- **Sections**: CS220-01/02 — 4 credit units
- **Regular meeting times**: M., Tu., W., Th.: Sec. 1, 8:50-9:45am; Sec. 2, 9:55-10:50am
- **Lecture room**: Centennial 2205
- **Lab days and room**: Some Mondays and some Wednesdays will be lab sessions, held in 16 Wing.
  - See the course website for each week’s plan.
- **Course website**: [https://cs.uwlax.edu/~jmaraist/220-fall-19](https://cs.uwlax.edu/~jmaraist/220-fall-19)
  - Contains the up-to-date calendar for the next few weeks of deadlines and the lecture/lab schedule, plus other essential references
- **Other pages to bookmark for this class**:
  - **Autolab — for Java assignment submission** [https://euryale.cs.uwlax.edu/courses/cs220-fa19-jmaraist](https://euryale.cs.uwlax.edu/courses/cs220-fa19-jmaraist)
    - You must be on campus, or using the UWL VPN, to access this site,
  - **Links to course documents (on Canvas)** [https://uwlac.instructure.com/courses/217854/pages/cs220-class-resources](https://uwlac.instructure.com/courses/217854/pages/cs220-class-resources)
    - Includes links to the full syllabus and course pack of lecture slides, exercises, etc.
  - **Available online quizzes (on Canvas)** [https://uwlac.instructure.com/courses/217854/quizzes](https://uwlac.instructure.com/courses/217854/quizzes)
- **Prerequisites**: CS120
- **Corequisites**: CS225 recommended
- **Catalog description**: This is a second course in the design of programs. Emphasis is placed on data abstraction and its application in design. Definitions of abstract data types are examined. The following structures are examined as methods for implementing data abstractions: recursion, sets, stacks, queues, strings, and various linked lists. Students will be expected to write several programs using these techniques in a modern programming language.

- **Instructor's name**: Dr. John Maraist
  - Vocative “Dr. Maraist” or “Prof. Maraist”, pronouns he/him/his
Office location 209 Wing Technology Center

Email jmaraist@uwlax.edu

About http://cs.uwlax.edu/~jmaraist

Office hours and appointments My office hours and appointment availability are listed on the course website. To make an appointment, ask by email at least one school day ahead of time.

References ZyBook Java text; Java: A Beginner's Guide, Herbert Schildt, Oracle Press; see the course website for other resources

0.2.2 Key dates

Tuesday, September 3 First class
Thursday, September 5 Midterm 0
Tuesday, September 24 Tentative date of Midterm 1
Monday, October 14 Tentative date of Midterm 2
Tuesday, October 29 Tentative date of Midterm 3
Tuesday, November 19 Tentative date of Midterm 4

I will confirm the actual midterm test dates at least two weeks beforehand.

November 21-22 Thanksgiving break

Saturday, December 14, 12:15-2:15pm Final exam, Section 220-01
Tuesday, December 17, 2:30-4:30pm Final exam, Section 220-02

• The final exam dates and times are set by the university; the schedule is available at www.uwlax.edu/records/faculty-staff-resources/final-exam-schedule . Per Note 1 on that page, our class is considered a MWF class for purposes of final exam scheduling. Do not plan to leave for holiday travel until after the exam date for your section of this class.

0.2.3 The objectives and outlook of this class

This class offers the opportunity to master the fundamentals of software development. We will use the Java programming language, but the skills we will convey are applicable to most programming and scripting languages in use today. Over the course of the semester, we will deepen your understanding of topics related to software development, including problem solving techniques, fundamental programming constructs, and their application to algorithm design and to the Java programming language.

In CS120 we learned about the design of simple algorithms and their implementation as Java programs executed as a single, sequential thread. We began with core elements of imperative programming: variable assignment, use and update; expressions, boolean logic, selection and iteration, subroutine use, arrays; and the language issues of syntax, declarations, scope, and subroutine creation and invocation. We introduced notions of object-oriented programming: classes and objects, constructors and methods, and inheritance. In CS220 we will build further on these foundations, including:

• Designing, implementing and understanding programs using object-orientation (including inheritance, overloading, overriding, and method polymorphism); recursion; generics; single- and multi-dimensional arrays; the separation of abstraction and implementation; exceptions; the local file system; and elementary data structures.

• Demonstrating understanding of Java’s use of your computer’s memory for method calls, variable and parameter storage, references, objects and arrays.

• Analyzing the asymptotic complexity of simple algorithms and presenting the result in big-\(O\) notation. We will focus in particular on a number of searching and sorting algorithms for arrays, and on operations on linked lists.
• Through the semester we will learn to debug programs, that is, to fix a range of problems, including infinite
loops and various exceptions.

• Modern programming system make heavy use of standard pre-programmed libraries, so our work will use a
number of external libraries, including for data structures and file I/O.

• Communication is as essential in computer science as in any other field. Although we will not have larger-scale
writing exercises, I do require and will assess program comments, and examinations will include short-answer
questions.

• Any local culture of programmers (such as a workplace or a community project) will adopt or be assigned
stylistic conventions for programs and for comments. We will point out and follow a number of these guidelines,
as well as general professional habits.

This class is focused tightly on mastering a specific set of skills, and on the knowledge associated with those skills.
Mastering any new mental or physical skill requires practice and discipline. You should plan to spend an average
of about ten to twelve hours a week (not counting our class meetings) preparing for class, working assignments,
and otherwise studying or practicing class material. As with a sport or musical instrument, you will not develop
programming skills without committing serious and regular effort to actually programming.

The focus on skills, and the elementary nature of the material we cover, means that this class is highly cumulative.
Topics from later in the class rely very heavily on earlier topics. Everything in this class relies very heavily on the
material of CS120. Even where assignments and exams focus on particular later topics, it is unavoidable that earlier
topics will be essential components of later work.

The skills you will gain in this class will generally fall into one of these four categories:

1. **Designing an algorithm**

2. **Writing programs or parts of programs**

3. **Constructing and debugging correct executable programs**

4. **Analyzing programs and code** to accurate predict how it will behave

It is important to master all of these skills over the course of the semester, but we recognize that some people take longer
to master some aspects of algorithmic thinking and programming. So when computing final grades, I will replace
earlier grades from a particular skill category with the weighted average of later grades from the same category. The
Assessment section of the syllabus details exactly how this calculation will work. (Miscellaneous and administrative
assignments will have separate categories for grading purposes.)

### 0.2.4 Assessment

For assessment, we divide the assignments and examinations into the following phases: *preparatory homework* (which
will include the quizzes), *supplementary homework*, *labs*, *projects*, *mid-term exams* and the *final exam* (in that order).

Each marked item will be attributed to one category including: *programming* (which includes book work, labs, projects
and most other independent work), *algorithm design*, *describing how code will execute*, *conveying understanding of
concepts*. The latter three categories are for exams; the first category is for out-of-class homework and projects. The
points of items for a particular category in each phase of the class will be adjusted to be no less than the weighted
average of items of the same category in the next later phase of the class. So for example, your percentage score on a
midterm exam asking you to predict the effect of some piece of code will not be less than the weighted average of your
scores on the final exam questions asking you to predict the effect of code. This adjustment will be transitive from the
final exam backwards.

#### Forms of assessment and their weight

Your grade for each class assignment and phase will be calculated as a weighted average. In turn your final grade will
be the weighted average of the assessment of your work, adjusted as described above, and weighted as follows:

- *Preparatory homework and quizzes*: 5%

---

8
Supplementary homework: 10%

Preparatory homework is designed to let you come to class ready to engage with the day’s material. Sometimes it will review earlier concepts (from CS120, or from previous topics in this class); sometimes it will ask you to experiment with aspects of Java which we will consider in more detail that day. Supplementary homework follows up on class topics to reinforce what we discuss. The supplementary problems help you identify aspect of the new material which you may not yet understand, and prepare you for labs and projects.

Labs: 7%

Projects: 25%

Labs and projects ask you to apply the material you learn in class and practice in homework. Lab assignments tend to be shorter, and in some labs I will guide you through parts of the solution. In labs we will also focus on adhering to the discipline of programming which we adopt for this class. Labs will be graded qualitatively on your progress in class, and demonstrated in the submitted work product, towards mastering the exercised skills. Projects, especially later projects, require you to write longer and more substantial programs, often with less structure provided than in the labs.

Midterm tests: 25%

The midterm tests each cover a particular set of knowledge. They are not explicitly cumulative, in the sense that their questions will focus on the topics for that test, and not on past topics. However, the material of this course is highly cumulative in nature: so you may need to use the skills from past topics in order to answer questions correctly. For example, answering questions about arrays will require the ability to write loops, even though loops were a topic for CS120.

The topics tested on the midterms are as follows:

– Midterm 0: Topics from the prerequisite course CS120. Appendix C of the course pack contains a list of the department’s learning outcomes for CS120; all of these topics are in scope for this test except GUI programming.
– Midterm 1: The stack-and-heap model of program execution, recursion, asymptotic complexity, exceptions (Section 1 of the course pack).
– Midterm 2: Classes and objects (Section 2 of the course pack).
– Midterm 3: Arrays (Section 3 of the course pack).
– Midterm 4: Linked lists, stacks, queues (Section 4 of the course pack).

In calculating the share of each midterm towards your final grade, I will weigh the last four midterms evenly, and will weigh Midterm 0 at half the weight of each of the others.

Final examination: 25%

The final examination will be cumulative over the topics we cover.

Participation and professionalism: 3%

Partial credit for programs, whether on examinations or projects, may be awarded only for programs whose design is documented via comments in the manner expected under our program design discipline.

Final grades

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>95.0 ≤ g &lt; 100.0</td>
<td>A</td>
</tr>
<tr>
<td>90.0 ≤ g &lt; 95.0</td>
<td>AB</td>
</tr>
<tr>
<td>84.0 ≤ g &lt; 90.0</td>
<td>B</td>
</tr>
<tr>
<td>78.0 ≤ g &lt; 84.0</td>
<td>BC</td>
</tr>
<tr>
<td>70.0 ≤ g &lt; 78.0</td>
<td>C</td>
</tr>
<tr>
<td>60.0 ≤ g &lt; 70.0</td>
<td>D</td>
</tr>
<tr>
<td>Lower results do not earn a passing grade. In addition, to get a final grade above C, you must pass the final exam. The university uses annotated F grades for cases of failure with cessation of class activity and attendance; where such grades are appropriate I will draw them from both the assignment results and attendance records.</td>
<td></td>
</tr>
</tbody>
</table>
Feedback

Formative assessments are those whose purpose includes giving feedback to you, and shaping your learning. You will receive feedback on all formative assessments, and are expected to use that feedback to improve your future performance. Other work is considered summative, intended not for feedback or as learning tools but only as measurements of skill. You will not receive detailed feedback on summative assessments.

- Programming work due before Thanksgiving, as well as the first four midterm tests, are all considered formative. The last midterm will treated as formative if time allows.

  Feedback on lab work will often be delivered in person in the lab session; it is your responsibility to ask questions during labs when you are unable to complete the assignments. Feedback on programming assignments will come not only from me directly, but also from the Autolab submission system: you are responsible for reading the output of that system when you submit work to see where it finds incorrect or unimplemented aspects of your code.

- Later work is considered summative. The final project and the final examination are summative assessments, as well as programming work due after Thanksgiving (although I will return as much of it as early as possible). The last midterm may be deemed summative if time requires.

0.3 Acknowledgments

Much of this document is from material shared among the various professors who have taught this class in recent years, including Kenny Hunt, Elliot Forbes, Allison Sauppé, and in particular Joshua Hurley and Jason Sauppe. Some of the exercises are derived from other sources, and individually attributed.
1 The stack and the heap (although for the moment, mostly just the stack)

1.1 How Java organizes your computer’s memory

You know that your program uses the computer’s memory

- It uses memory for local variables
- It uses memory to pass parameters to methods
- It uses memory to return values from methods
- It uses memory to keep track of where we are in one method when we call another
- It uses memory to create arrays
- It uses memory to create objects of a class, and for the objects’ fields

Our programs use memory constantly
But how does Java manage it all? How does it organize memory to it all possible?

The stack and heap
At the highest level, there are two main divisions of your computer’s memory: the stack and the heap

- The space in between the stack and heap is unused, or free memory
- Both the stack and the heap can grow and shrink
  - But they are each kept to their own end
  - They grow towards the middle, shrink towards their end
- When the stack and heap collide, it means that your program has run out of memory
  - So the system terminates it
  - Would look to you like any other program crash

Method calls on the stack
Every time your program calls a method, a stack frame is added to the stack

```java
public void f() {
    int a = 4;
    g(10); // Line 3
    // ... 
}

public void g(int x) {
    int b = 5;
    // ...
}
```
Some things each stack frame can contain:

• Storage for the parameters passed to the method
• Storage for the method’s local variables
• Temporary workspace for more complicated expressions and statements
• A note of what we should resume doing when this method ends
• The location of the previous stack frame

For efficiency, different systems may actually use other parts of the machine for some of these things — but this idea of stack frames is always useful for understanding our programs

**Returning from a method discards its frame**

When a method exits, Java does not keep its frame

• So the stack grows when we call a method, shrinks when we exit a method

• Methods are exited in the opposite order that we call them
  
  – This behavior is called *last in, first out*
  
  – Means that the stack is always a *contiguous* block of in-use memory

```java
public void f() {
    int a = 4;
    g(10);
    h(11, 4.5);
    // ...}
public void g(int x) {
    int b = 5;
    // ...}
public void h(int y, double z) {
    // ...}
```
Exercise 1.1.1. Draw pictures of the stack following a call to $f(3)$ immediately before $g$ is called, immediate before $g$ exits, and immediately after $g$ exits. What is printed, and when?

```java
public void f(int x) {
    int s=0;
    for(int i=1; i<=x; i=i+1) {
        s = s+i;
    }
    g(x,s);
    System.out.println(s);
}

public void g(int y, int z) {
    int s = 100-y;
    System.out.println(s);
    z = z+s;
    System.out.println(z);
}
```

Exercise 1.1.2. Draw the stack configuration at each call to the print statement of $h_2$ following an initial call of $h_1(3)$. What is printed each time?

```java
public void h1(int m) {
    // Code...
}
```
Hidden declarations

The stack of frames can also help us understand the use of $i$ in this Java:

```java
public void f(int x) {
    int i = 10;
    if (x>0) {
        int i = x+3;
        System.out.println(i);
        System.out.println(x);
    }
    System.out.println(i);
}

public void g(int y) {
    f(y);
}
```

- When Java first begins a call to $f(5)$ from $g(5)$, we have a frame with storage for $x$ and the first declaration of $i$
- When the if test succeeds, we enter a new scope for local names
  - The new scope has its own meaning for $i$, so the use of $i$ in the enclosing scope becomes hidden
  - The new scope knows that for $x$, we should still look in the enclosing scope
  - But both of these frames know not to look in $g$'s frame for a binding for $y$
- After the affirmative if block, when the nested scope goes away, Java simply removes its frame from the stack
Exercise 1.1.3.  Consider this method:

```java
public static int sumSquares(int a) {
    int result = 0;
    // Point [A]
    for(int i=0; i<a; i++) {
        result = result + (i*i);
        // Point [B]
    }
    // Point [C]
    return result;
}
```

Sketch the configuration of the stack and heap during the execution of a call to `sumSquares(2)`:

- When point A is reached
- Each time point B is reached
- When point C is reached

Exercise 1.1.4.  Consider this method:

```java
public static int sumSquaresAndCubes(int a) {
    int result = 0;
    for(int i=0; i<a; i++) {
        result = result + (i*i);
    }
    return result;
}
```
for(int j=0; j<a; j++) {
    result = result + (j*j*j);
    // Point [A]
}
return result;

Sketch the configuration of the stack and heap each time a call to `sumSquaresAndCubes(3)` reaches Point A.

**Exercise 1.1.5.** Consider these statements, where we assume that x and y are declared and set locally with type int:

```java
while (x>10) {
    int y = x*x;
    System.out.println(x + " " + y);
    x = x-y;
}
```

Run these statements in your Java environment if you are unsure what values they print. Sketch the configuration of the stack and heap each time the `println` statement is reached.

**Exercise 1.1.6.** Consider these statements, where we assume that x, y and z are declared and set locally with type int:

```java
while (x >10) {
    if (x % 2 == 0) {
        int y = x*x;
        System.out.println(x + " * " + y);
        x = x-y;
    } else {
        x = x-1;
    }
}
```

Run these statements in your Java environment if you are unsure what values they print. Sketch the configuration of the stack and heap each time the `println` statement is reached.

**Exercise 1.1.7.** Consider these statements, where we assume that x, y and z are declared and set locally with type int:

```java
while (x>10) {
    while (y>5) {
        int z = x*x;
        System.out.println(x + " * " + y + " * " + z);
        y = y - z*z;
    }
    System.out.println(x + " * " + z);
    x = x - z*z;
}
```

Run these statements in your Java environment if you are unsure what values they print. Sketch the configuration of the stack and heap each time a `println` statement is reached.

**Where do objects live?**

When a method exits, objects it creates may still be in use

```java
public void f1() {
...
```
Red c = new Red();
f2(m);
f3();
// ...
}

public void f2(Red m) {
    Green x = new Green();
    m.setG(x);
}

public class Red {
    private Green g;
    public void setG(Green g) {
        this.g = g;
    }
}

What happens if we store the Green object on the stack?

Stack storage is not appropriate for object instances

Objects live in the heap
We store objects on the other end of memory, away from the stack’s expansion and contraction
Heap management

- Consecutive objects may have variable and unpredictable lifespans
- So free spaces can open up inside the heap
- The heap must be more sophisticated than a simple stack, so that we can track and recapture these free fragments

Contents vs. locations
There is an important distinction between the contents of a heap-allocated object and its address

- The address is like a library book call number
- Local variables actually store the address
  - Initially null
- Java passes the address as a parameter

Exercise 1.1.8. Use println to see what is initially stored in uninitialized local variables with the following types: int, long, double, boolean, String, and Red (as defined on p. 15 above).

```java
public class Green {
    private final int brightness;
    public Green(int brightness) {
        this.brightness = brightness;
    }
    public int getBrightness() { return brightness; }
}
```

Exercise 1.1.9. For the definition of Red on p. 15 above, and this definition of Green,

```java
public class Green {
    private final int brightness;
    public Green(int brightness) {
        this.brightness = brightness;
    }
    public int getBrightness() { return brightness; }
}
```

sketch the state of the stack and heap after each statement in this sequence:

Red r1 = new Red();
Red r2 = new Red();
Green g1 = new Green(100);
r1.setG(g1);
Green g2 = new Green(7);
r1.setG(g2);
1.2 Recursion

What happens with this code?

What happens when we call `start()`?

```java
public static void start() {
    int x = selfCall(5);
    System.out.println("Fin: " + x);
}
public static int selfCall(int x) {
    System.out.println("Pre: " + x);
    return selfCall(x);
}
```

- Like `printGreater`, `selfCall` calls a method inside itself
- Unlike `printGreater`, `selfCall` calls itself

Recursion is when a thing is defined in terms of itself

- In programming, it occurs when a method calls itself

But there is a problem with this use of recursion

- What happens when `main` calls `start()`?

Components of recursion

For recursion to work properly, we need:

**Recurrence condition** Cases where the code will call itself to generate repetition

**Base case(s)** Some point that we are guaranteed to reach, where the recurrence will stop and the method will not call itself anymore

Just as for a loop!

- Loops need a continuation condition
- Which eventually becomes false, and makes us exit the loop

**Exercise 1.2.1.** Trace the following recursive method calls to determine the output of `start`:

```java
public static void start() {
    int x = selfCall(5);
    System.out.println("Fin: " + x);
}
public static int selfCall(int x) {
    System.out.println("X: " + x);
    if (x <= 0) {
        System.out.println("Base case!");
        return 0;
    } else {
        int r = x + selfCall(x - 1);
        System.out.println("R: " + r);
        return r;
    }
}
```
public static void start() {
    System.out.println("A = " + mystery(2,2));
    System.out.println("B = " + mystery(3,4));
    System.out.println("C = " + mystery(10,7));
}

public static int mystery(int a, int b) {
    if (b == 0) {
        return 0; // A
    } else if (b == 1) {
        return a; // B
    } else {
        return a + mystery(a, b-1); // C
    }
}

What happens with this version of selfCall (Exercise 1.2.1)?

public static void start() {
    int x = selfCall(5);
    System.out.println("Fin: " + x);
}

public static int selfCall(int x) {
    System.out.println("X: " + x);
    if (x <= 0) {
        System.out.println("Base case!");
        return 0;
    } else {
        int r = x + selfCall(x - 1);
        System.out.println("R: " + r);
        return r;
    }
}

How do we use recursion?

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— Niklaus Wirth, Algorithms + Data Structures = Programs

Many mathematical series are defined by recursive recurrence relations

• Factorial: \( n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \)

  \( \text{– fact}(0) = 1 \)
- fact(n) = n \cdot \text{fact}(n-1)

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, \ldots

- \text{fib}(0) = 0
- \text{fib}(1) = 1
- \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)

Exercise 1.2.3. Using the formulas above, write a class RecursiveFactorial with a single recursive static method \text{factorial} which, given a non-negative integer \( n \), returns \( n! \), the factorial of \( n \). Give the argument the type \text{int}, but give the result the type \text{long}. Why are these the different types appropriate? □

Exercise 1.2.4. Using the formulas above, write a class RecursiveFibonacci with a single recursive static method \text{fib} which, given a non-negative integer \( n \), returns the \( n \)th Fibonacci number. As with factorial, give the argument the type \text{int}, but give the result the type \text{long}. What do you notice about the time it takes for your method to run as the value of \( n \) increases? How does this change compare with RecursiveFactory? □

Exercise 1.2.5. Write a class RecursiveMersenne with a single recursive static method \text{mersenne} which calculates the \( n \)th Mersenne number, defined using the recurrence

\[
\text{f}(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2 \cdot \text{f}(n-1) + 1 & \text{otherwise}
\end{cases}
\]

Trace the state of the stack at each call to \text{mersenne}(4). □

Exercise 1.2.6. Write a class PalindromeCheck with an overloaded static method \text{isPalindrome}. You should overload \text{isPalindrome} with a one-argument version of the method, and a three-argument version of the method. The one-argument method should

- Take a single \text{String} argument, and
- Return \text{true} exactly when the argument string is a palindrome, that is, the same string forwards and backwards.

You should implement the one-argument method as a single call to a three-argument method which you implement using recursion. The three-argument version should:

- Take two integers and a string: the two integers mark the next two positions within the string to be checked to see if they are the same, and
- Also return \text{true} exactly when the argument string is a palindrome.

Exercises 1.2.7 and 1.2.8 are about two different ways of finding the greatest common divisor of two positive integers \( a \) and \( b \): the largest number which divides both \( a \) and \( b \) with zero remainder. All pairs of positive integers have 1 as a common divisor, since every number can be easily divided by 1, but challenge of finding the greatest common divisor is to avoid a brute-force solution involving repeated trial divisions. The first solution dates to antiquity from the \textit{Elements}, attributed to Euclid of Alexandria. The second solution is by Edsger Dijkstra from the 1960s, and favors subtraction over division in its calculations, since subtraction was sufficiently faster than division at that time to make Dijkstra’s approach preferable.
Exercise 1.2.7. Write a class `Gcd` with a single recursive method `gcd` which calculates the greatest common divisor of its two `int` arguments using Euclid’s classical method: to find GCD(a,b) where \( a \geq b \):

1. If \( a=0 \), then GCD(a,b)=b
2. If \( b=0 \), then GCD(a,b)=a
3. Find \( q,r \) such that \( a=bq+r \)
4. Find GCD(b,r)
   • Since GCD(a,b)=GCD(b,r)

Trace the state of the stack at each call to `gcd(192,78)`.

Exercise 1.2.8. Write a class `Gcd` with a single static recursive method `gcd` which calculates the greatest common divisor of its two `int` arguments using Dijkstra’s method: to find GCD(a,b)

1. If \( a=b \), then GCD(a,b)=a
2. If \( a>b \), then GCD(a,b)=GCD(a-b,b)
3. Else GCD(a,b)=GCD(a,b-a)

Your method’s result should also be an `int`. Trace the state of the stack at each call to `gcd(192,78)`.

Exercise 1.2.9. Sierpinski’s carpet is one kind of fractal image — an image which recurs in smaller form inside of itself. For this exercise you should use the `DrawingPad` class discussed in Section D.5.

Drawing a Sierpinski fractal follows these steps:

1. Start with a square drawing region
2. Divide the square into 9 sub-squares
3. Color the center sub-square

4. Repeat the process for each of the remaining 8 sub-squares and so, as far as desired.

Write a class `PartialSierpinski` which uses a recursive method to draw \( n \) levels of Sierpinski’s Carpet in a new graphics window. The main method of your class should:

- Query the user for the size of window to draw, and the number of iterations it should make.
- Create a square `DrawingPad` whose length and width are the size given by the user.
- Call your recursive method to draw Sierpinski’s carpet to the given depth.

**Exercise 1.2.10.** Write a class `FullSierpinski` which also uses a recursive method to draw \( n \) levels of Sierpinski’s Carpet. Instead of querying for the depth of recursion, `FullSierpinski` should cease making recursive calls when the width of a sub-square is below three pixels.

1.3 **Discussing execution time**

**Measuring running time**

How should we measure the running time of a program? And how precise do we need to be?

- Wall clock time, CPU time
- Instruction count: Java instructions or machine-level instructions
- Number of basic units of work executed
  - What’s relevant as a "unit of work" might vary from algorithm to algorithm
  - Must require (at most) some *constant*, small time
  - Must be able to express the work of the algorithm in terms of these units
Units of work

**For factorial**
For example, with factorial the quantum of work includes:

1. Test to see if we have a base case
2. Maybe set up a method call and result storage
3. Maybe perform a multiplication

**For Fibonacci**
What is in one quantum of work for Fibonacci?

1. Test to see if we have a base case
2. Maybe set up method calls and results storage
3. Maybe perform an addition

**Exercise 1.3.1.** Given an input value \( n \), how many times will \( \text{RecursiveFactorial.factorial} \) repeat its quantum of work?

**Exercise 1.3.2.** Given an input value \( n \), how many times will \( \text{RecursiveFibonacci.fib} \) repeat its quantum of work?

**Exercise 1.3.3.** There are many folk songs similar to "The Twelve Days of Christmas," where each successive verse increases in length by a certain amount. Let us assume that someone’s crazy aunt insists that we all sing one of these songs where the first verse has \( F \) words, and that each verse increases in length by no more than \( W \) words. If the crazy aunt insists on \( n \) verses, what is the maximum number of words which we might be forced to sing?

*Adapted from [2, p. 69]*

**Exercise 1.3.4.** Consider a class \( \text{Point} \) representing points on the plane, which has a method \( \text{dist} \) for finding the distance between this point and another,

```java
public int dist(Point that)
```

The following method finds the shortest distance between any two elements of an array of \( \text{Point} \) instances. What is the quantum of work for this method?

```java
public static int shortestDist(Point[] points)
{
    int result = Integer.MAX_VALUE;
    for(int i=0; i<points.length; i=i+1) {
        for(int j=i+1; j<points.length; j=j+1) {
            int dist = points[i].dist(points[j]);
            if (dist < result) {
                result = dist;
            }
        }
    }
    return result;
}
```

Given an array of \( n \) points, how many times does the method execute the quantum of work? *Adapted from [2, p. 51]*
We do now have formulas

We do now have formulas for the quanta of work required for different algorithms, but these formulas are complicated. What if we are using these formulas to discuss the resource needs of much larger, more complicated programs using these methods?

- In this form, our assessment of runtimes are too complicated to be useful

Inessential details

- We generally do not need to worry about small cases
  - The special cases for small values are not part of any "big picture" of how the methods behave as the input gets large
- When we add terms of a formula together, polynomial terms of smaller degree become uninteresting at large values
  - Think about a formula \(f(x) = x^4 + x\)
  - At large values of \(x\), the \(x^4\) part is overwhelmingly bigger than the \(x\) part
- We are also not very interested in constant coefficients
  - Although \(f(x) = 2x\) is certainly a faster-growing function than \(f(x) = x\)
  - We are measuring the speed of computer programs, and a system which is twice as fast as our current one is not unrealistic

Big-O notation

Intuitively, we could just drop the details in which we are not interested

But in science, engineering and math, informal intuition is not good enough — we need a clear and rigorous reasoning framework

- **Big O notation** is a mathematical notation used to describe the limiting behavior of a function as its input grows

  For a function \(f(n)\), we say that \(f(n)\) is \(O(g(n))\) for some function \(g\) if and only if there exists a positive constant \(M\) and value \(n_0\) such that

  \[|f(n)| \leq M|g(n)|\] for all \(n \geq n_0\).

- This is written as \(f(n) = O(g(n))\) or \(f(n) \in O(g(n))\).

  When \(f(n)\) is \(O(g(n))\), it means that \(g\) gives an **asymptotic upper bound** for \(f\)

Applying big-\(O\)

When we discuss the running time of a program, we will describe the order of the running time via big-O notation

- A program which runs in a constant amount of time for any input has order \(O(1)\)
- If a program with *linear* runtime, that means that its running time is in \(O(n)\) for input of size \(n\)
  - Similarly *quadratic* runtime for \(O(n^2)\), *cubic* runtime for \(O(n^3)\), and so on
- **Logarithmic** programs have runtimes which are \(O(\log n)\)
  - In between constant and linear
  - The base does not matter
- **Exponential** programs have runtimes which are \(O(a^n)\) for some \(a > 1\)
We can show that the quantum counter for our recursive implementation of \texttt{fibonacci} has $O(2^n)$.

- The base does matter here — $O(3^n)$ is much worse than $O(2^n)$.

- When we use big-\(O\) notation this way, we say that we are describing the \textit{asymptotic complexity} of the program.

We will see examples of programs with most of these orders over the course of the semester.

**Exercise 1.3.5.** Using your result from Exercise 1.3.1, find the best big-\(O\) bound for the amount of work done by \texttt{factorial} for input value \(n\).

**Exercise 1.3.6.** Using your result from Exercise 1.3.4, find the best big-\(O\) bound for the steps taken by method \texttt{shortestDist} for an array of size \(n\).

### Problems vs. algorithms

So far we have discussed the asymptotic complexity of \textit{one particular implementation} of a problem.

- We can also discuss the asymptotic complexity of a \textit{problem itself}.
  - When we discuss the complexity of a problem, we mean that \textit{any} implementation will have a complexity which is \textit{at least} that bad.
  - Fibonacci is a great example here.
    - We started discussing complexity to understand why our recursive implementation was so slow.
    - That implementation is exponential — but the problem of finding Fibonacci numbers is not.

### 1.4 Improving recursion

#### Fibonacci with a loop

How can recursive Fibonacci match the performance of the loop-based version?

```java
public static long fib(final int num) {
    long fib=0;
    long next=1;
    for(int n=0; n<num; n++) {
        final long newNext=fib+next;
        fib = next;
        next = newNext;
    }
    return thisFib;
}
```

- The loop runs (about) \(n\) times, so we \textit{should} be able to find a way that recurs \(n\) times.
- Note how we use next and fib:
  - Both calculated on each pass through the loop.
  - Both preserved from one pass through the loop to the next.
- How can we provide both from one recursive call to the next?
  - By passing both as parameters!
Recursion with accumulating parameters

- Instead of calculating the result in a method body after the return of a recursive call,
- Calculate the result in the arguments of the call

```java
static long fibHelper(int n, long fib, long next) {
    if (n<1) {
        return fib;
    } else {
        return fibHelper(n-1, next, fib+next);
    }
}
```

```java
public static long fib(int n) {
    return fibHelper(n, 0, 1);
}
```

**Exercise 1.4.1.** An anagram is the rearrangement of the letters in a word or phrase. Sometimes anagrams can be funny, or at least coherent: for example, one anagram for admirer is married. Other anagrams make no particular sense: for example, one anagram for blender is deerlnb.

Write a class Anagrammer with:

- A static method `printAnagrams` which takes a single `String` argument, returns no result, and prints (one per line) all of the anagrams of its first argument.
- A second static helper method called by `printAnagrams`, which is recursive, and which accepts additional helper parameters as needed.

### Towers of Hanoi

Given a set of six disks of varying sizes, stacked on poles:

- Move the disks from the left pole to the right pole, while ensuring that only one disk is moved at a time and a large disk is never placed on top of a smaller one.

Solving the Towers of Hanoi puzzle

The start of a strategy:

- Move the smallest five disks from the left pole to the center pole
- Move the largest disk from the left pole to the right pole
- Move the smallest five disks from the center pole to the right pole
Identifying subproblems
Until we move the smallest five disks to the center pole, we do not need to move the bottom one at all (nor can we).

- We can pretend the largest disk doesn’t exist, giving us a new smaller version of the same problem!
- Now we just have to figure out how to move the smallest five disks from the left pole to the center pole…

Solving the subproblem
To solve the subproblem, the same reasoning applies:

- Move the smallest four disks from the left pole to the right pole
- Move the second largest disk from the left pole to the center pole
- Move the smallest four disks from the right pole to the center pole

A simple recursive solution

```java
public static void moveDisks(final int numDisks,
                              final Pole source,
                              final Pole dest,
                              final Pole temp) {
    if (numDisks > 1) {
        moveDisks(numDisks-1, source, temp, dest);
```

```java
    } // end if
```

```java
    // move largest disk
    source.moveDisk(dest);
```
// Base case: move disk from source pole to dest pole
moveDisk(source, dest);

if (numDisks > 1) {
    moveDisks(numDisks-1, temp, dest, source);
}

Recursion and iteration

Every looping structure can be replaced with recursion and every recursive solution can be replaced with looping

• The basic while loop:

        while (condition) {
            loop body code;
            progress statement;
        }

• Recursive method:

        private void methodName() {
            if (condition) {
                loop body code;
                progress statement;
                methodName(); // recur (loop again)
            } else {
                return n * factRec(n-1);
            }
        }

Recursion or iteration?

Recursion and loops may be interchangeable, but sometimes a problem makes more sense with one or the other.

public static long factRec(final int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factRec(n-1);
    }
}

public static long factLoop(final int n) {
    long product = 1;
    for (int i=1; i<=n; ++i) {
        product *= i;
    }
    return product;
}

Recursive implementations can incur an overhead at runtime

• Space for keeping track of method calls and local variables
• Time for executing each method
Many compilers can optimize tail calls — when the argument of the return statement is just the recursive call — to be just as efficient as loops

```java
public static int fact(final int n) {
    return helper(n, 1);
}
public static int helper(final int n,
                            final long prod) {
    if (n<2) {
        return prod;
    } else {
        return helper(n-1, n*prod);
    }
}
```

For very small problems with an easy iterative solution, favor the loop

**Mutual recursion**

In some cases, recursion can be done indirectly and may not be immediately obvious.

- In mutual recursion, methods call each other

```java
public boolean isEven(int n) {
    if (n == 0) {
        return true;
    } else {
        return isOdd(n-1);
    }
}

public boolean isOdd(int n) {
    if (n == 0) {
        return false;
    } else {
        return isEven(n-1);
    }
}
```

- As usual, must make sure to move towards a base case!
- Understanding mutual recursion can be more subtle
  - But sometimes it’s the easiest solution

---

**Exercise 1.4.2.** Write a class Parity Checker with two static, mutually recursive methods `isOdd` and `isEven` (which it may assume to be non-negative), each taking one int argument, and not using the remainder operator %. Only one of these methods should distinguish base/recursive cases; the other should simply convert its present case to a call to the other with the same argument.
1.5 Exceptions

Errors in programming
There are two basic forms of programming error

Syntax errors
Due to a violation of the \textit{syntax} of the programming language

\begin{itemize}
\item Incorrectly typed code, misspellings, wrong punctuation
\item Results in a \textit{compile-time error}: code cannot be run
\end{itemize}

Logic errors
Arise from syntactically correct code that can compile and run but does not work as expected

\begin{itemize}
\item JVM will notice some errors when the code executes
  \begin{itemize}
  \item But \textit{not all} errors will be caught!
  \end{itemize}
\item May result in a \textit{run-time error}
\end{itemize}

Errors at runtime
When the JVM detects an error, it \textit{throws} an \textit{exception}

```java
public class Driver {
    public static void main(String[] args) {
        int j = 0;
        int k = 25/j;
    }
}
```
gives:

```
Exception in thread "main" java.lang.ArithmeticException
    at Driver.main(Driver.java:4)
```

\begin{itemize}
\item The exception is identified by \textit{type}: \texttt{ArithmeticException}
\item The message has a \textit{stack trace} with the active code (method, class, line number)
  \begin{itemize}
  \item Sometimes called a \textit{traceback}
  \end{itemize}
\end{itemize}

A longer stack trace

```
Exception in thread "main" java.lang.ArithmeticException
    at Thing.doSomething(Thing.java:9)
    at Thing.<init>(Thing.java:5)
    at Driver.main(Driver.java:3)
```

What does this stack trace tell us?

\begin{itemize}
\item The error itself happened at line 9 of method \texttt{doSomething()} of class \texttt{Thing}
\item Java started running \texttt{doSomething} because the constructor for class \texttt{Thing} (<\texttt{init}> called it from line 5 of file \texttt{Thing.java}
\item And in turn the \texttt{Thing()} constructor had been called from line 5 of the \texttt{main()} method of class \texttt{Driver}, file \texttt{Driver.java}
\end{itemize}
So many ways to fail!
Java uses different types of exceptions to describe different error conditions

Example 1

String str = null;
str.toLowerCase();

Throws NullPointerException

Example 2

double[] arr = new double[3];
arr[3] = 29.4;

Throws ArrayIndexOutOfBoundsException

Example 3

Scanner s = new Scanner(System.in);
int i = s.nextInt(); // and the input is non-numbers

Throws InputMismatchException

Throw your own
You can throw exceptions explicitly with a throw statement

throw EXPRESSION;

- Exceptions are just a particular kind of Java object
  - They all have superclass java.lang.Exception
- For example:

```java
public class SimpleFraction {
    private int numerator, denominator;

    public SimpleFraction(int n, int d) {
        if (d != 0) {
            numerator = n;
            denominator = d;
        } else {
            final IllegalArgumentException error = new IllegalArgumentException("Denominator is 0");
            throw error;
        }
    }
}
```

- IllegalArgumentException is also part of package java.lang

Failure is not instantaneous

- When we run a program or look at a stack trace, we get the impression that a throw statement simply ends a program
- But this is not actually true
- In fact, the failure happens in small steps — one block of code at a time
  - Because — as we will see now! — some blocks can recover from an exception
Block by block

- When an exception is thrown, the current block of code terminates immediately
- If the current block does not have code to handle the exception, then the JVM moves to the enclosing block of code
  - And then the next one
  - And so on
  - When the top-level of a method terminates, we return to the calling point of that method
- Each enclosing block or method terminates, one at a time, until either
  1. The top level of the program is reached (usually the main method)
     - At which point the program terminates and displays a run-time error message to the user
  2. Or a block of code that can catch and handle the exception is reached

Handling exceptions
The try/catch/finally statement can catch an exception

Statement syntax

```java
try{
    tryInstructionBody;
} catch (ExcClass1 parameterName) {
    exceptionHandlerBody1;
} catch (ExcClass2 parameterName) {
    exceptionHandlerBody2;
} finally {
    finallyBody;
}
```

- Exactly one try block
- Zero or more catch blocks
- The finally block is optional

Meaning
• Execution begins by running the body of try

• If an exception occurs, then:
  – Execution of the try block immediately stops
  – Java considers the catch clauses in order, looking for one whose declaration matches the actual exception type
  – “Matches” means that the exception is of the same class, or is a subclass (directly or indirectly), of the class declared in the catch

• If a finally clause is included, then it will always execute after the try and any catch clause

• If there was no exception, or if a catch clause handled the exception, then execution can continue after the try statement
  – Otherwise, the exception continues through the frames on the stack

Exercise 1.5.1.  What is printed when this code runs?

```java
try {
    String str = null;
    System.out.print(str.trim());
} catch (ArithmeticException e) {
    System.out.println("Math Error");
} catch (NullPointerException e) {
    System.out.println("No String");
} finally {
    System.out.println("Finished");
}
System.out.println("More code here");
```

Exercise 1.5.2.  What is printed when this code runs, and the user types in `bob`? When the user types in a valid number?

```java
final Scanner scanner;
final int x;
try {
    scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    x = scanner.nextInt();
    System.out.println("That was easy.");
} catch (InputMismatchException e) {
    x = 0;
} finally {
    System.out.println("Read " + x);
    scanner.close(); // Close scanner regardless
}
```

Exceptions and methods
The code that actually causes an exception does not always have to be directly inside a try block itself
Exception-causing code may be within another method that is called by the current one.

```
methodA() {
    try {
        methodB();
        ...  
    }
}

methodB() {
    try {
        methodC();
        ...  
    }
}

methodC() {
    ...
}
```

If `methodA` can catch the exception, it will, but otherwise the exception will be thrown back toward `main`, and may cause the program to fail.

If `methodB` can catch the exception, it will, but otherwise the exception will be thrown back to `methodA`.

Without a try/catch of its own, any exceptions in `methodC` get thrown back to `methodB`.

**Exceptions in normal code (no try)**

![Flowchart showing exception handling in normal code](image)

- **Yes** to Exception? leads to "Stop execution Are we in the outer-most code block?"
  - Yes: "Fail: throw exception from JVM, send error message with stack trace and other details"
  - No: Continue program normally

- **No** to Exception? leads to "Pass exception to next nesting code block, or to the method that called current one if no nesting remains"

**Exceptions in a try block**

![Flowchart showing exception handling in try blocks](image)

- **Yes** to Exception? leads to "Stop try. Matching catch exists?"
  - Yes: Pass exception up. Repeat at next level (try or not)
  - No: Execute finally

- **No** to Exception? leads to "Execute try. finally exists?"
  - Yes: Execute catch. finally exists?
    - Yes: Execute finally
    - No: Continue program normally
  - No: Continue program normally
• Caught exceptions can be re-thrown by handlers

• finally clause will still execute after re-throwing

```
try {
    String str = null;
    System.out.print(str.trim());
} catch (ArithmeticException e) {
    System.out.println("Math Error");
} catch (NullPointerException e) {
    System.out.println("No String");
    throw e;
} finally {
    System.out.println("Finished");
}
System.out.println("Never reached");
```

• `str` is declared but not initialized, so `trim` causes an exception

• Exception does not conform to `ArithmeticException`
  – So first catch block is skipped

• Exception does conform to `NullPointerException`
  – So message is printed
  – And then the exception is re-thrown

• Although we cancel execution of the handler(s) once the exception is re-thrown, still execute the finally block and print its message

• The last `println` never executes

Types of exceptions

When deciding what a try/catch block can and cannot catch, the JVM checks whether the actual exception conforms to the catch block’s named type

• There are many more kinds of exceptions than those shown below
Exercise 1.5.3. What does Java print as output when running method `topTry`?

```java
public void topTry() {
    try {
        System.out.println("try: Before methodA()");
        methodA();
        System.out.println("try: After methodA()");
    } catch (Exception e) {
        System.out.println("Handled by topTry!");
    }
    System.out.println("try: Finished");
}
```

```java
public void methodA() {
    try {
        System.out.println("A: Before methodB()");
        methodB();
        System.out.println("A: After methodB()");
    } catch (NullPointerException e) {
        System.out.println("Handled by methodA!");
    } finally {
        System.out.println("A: Finally");
    }
    System.out.println("A: Finished");
}
```

```java
public void methodB() {
    System.out.println("B: Throwing an exception");
    throw new ArithmeticException();
    System.out.println("B: Threw an exception");
}
```

Exercise 1.5.4. What does Java print as output when running method `tryMethods`? (This is not the same code as for Exercise 1.5.3; notice the different argument to the `throw` statement.)

```java
public void tryMethods() {
    try {
        System.out.println("try: Before methodA()");
        methodA();
        System.out.println("try: After methodA()");
    } catch (Exception e) {
        System.out.println("Handled by tryMethods!");
    }
    System.out.println("try: Finished");
}
```

```java
public void methodA() {
    try {
        System.out.println("A: Before methodB()");
        methodB();
        System.out.println("A: After methodB()");
    } catch (NullPointerException e) {
        System.out.println("Handled by methodA!");
    }
}
```
```java
public void methodB() {
    System.out.println("B: Throwing");
    throw new NullPointerException();
    System.out.println("B: Threw an exception");
}
```

**Exercise 1.5.5.** What gets printed by the following code? What does this tell us about the order of exceptions and super/subclass relationships among the `catch` clauses?

```java
try {
    String str = null;
    String lower = str.toLowerCase();
} catch (Exception e) {
    System.out.println("Generic exception");
} catch (RuntimeException e) {
    System.out.println("Runtime exception");
} catch (NullPointerException e) {
    System.out.println("Null pointer");
}
```

**Type hierarchy of exceptions**

- **Error**
  - OutOfMemoryError
  - StackOverflowError

- **Exception**
  - RuntimeException
    1. ArithmeticException
    2. ClassCastException
    3. IllegalArgumentException
    4. IndexOutOfBoundsException
    5. NullPointerException
    6. SecurityException
  - IOException
  - ...
Checked and unchecked exceptions

Some exceptions are checked, others are unchecked

- Checked exceptions must be caught and handled within the program
  - Enforced by the compiler
- Unchecked exceptions do not need to be explicitly handled in code
  - But will still cause runtime failure if they are not handled

Dealing with checked exceptions

Any code that might produce a checked exception must either:

- **Catch it**
  - Potential offending instructions placed inside a `try` block
  - Via a `catch` handler that matches the exception type
- **Propagate it**
  - Declare that this method can produce unhandled exceptions
public void writeToFile() throws IOException {
    // Code here that may produce an IOException
    // No try block is necessary
}

The throws declaration forces caller of the writeToFile to either catch the exception, or also propagate it.

Catch or propagate

Catch

    public void methodA() {
        try {
            writeToFile();
        } catch (IOException e) {
            // Code to handle the exception
        }
    }

Propagate

    public void methodB() throws IOException {
        writeToFile();
    }

Easy and wrong ways out

• A poor way to catch exceptions:

    public class MyClass {
        public static void main(String[] args) {
            try {
                // Exception-throwing code here
            } catch (Exception e) {
                // Do nothing
            }
        }
    }

• A poor way to propagate exceptions:

    public class MyClass {
        public static void main(String[] args) throws Exception {
            // Exception-throwing code here
        }
    }
Printing the stack trace

Sometimes we need to catch an exception but there is no graceful solution.

- *Print* the stack trace
- *Stop* the program

```
try {
    ...
} catch (NullPointerException e) {
    System.out.println("Invoking null pointer handler...");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Invoking index handler...");
} catch (OutOfMemoryError e) {
    System.out.println("Invoking memory handler...");
} catch (Exception e) {
    // Unsure how to resolve...
    e.printStackTrace();
    System.exit(-1); // Stops the program
```

Exercise 1.5.6. What does the following code print?

```
try {
    String str = null;
    System.out.print(str.trim());
} catch (NullPointerException e) {
    System.out.println("No String");
    throw e;
} catch (Exception e) {
    System.out.println("Generic exception");
} finally {
    System.out.println("Finished");
}
System.out.println("Reached?");
```

Exercise 1.5.7. What does the following code print?

```
public void test() {
    try {
        try {
            try {
                System.out.print("If at first you ");
                trying();
            } catch (ArithmeticException e) {
                System.out.print("do ");
            } finally {
                System.out.println("succeed");
            }
        } catch (NullPointerException e) {
            System.out.print("try ");
            throw e;
        }
        catch (NullPointerException e) {
            System.out.print("try ");
            throw e;
```

41
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException(String msg) {
        super(msg);
    }
}

Exercise 1.5.8. Create a class NoSuchFactorial with an object method getValue() returning an int. Update your factorial methods of both class FactorialFinder (Exercise D.2.17) and class RecursiveFactorial (Exercise 1.2.3) to throw a NoSuchFactorial exception when there is no factorial of its argument. Your class should certainly extend RuntimeException, but are there subclasses of RuntimeException in the standard Java library which might be a more suitable superclass for NoSuchFactorial?

Exercise 1.5.9. Repeat Exercise 1.5.8 for Fibonacci and Mersenne numbers.
2 Classes and objects

2.1 Classes, objects and inheritance

Software objects

A software object is an entity in a program that possesses state, attributes, and behaviors (actions)

• May interact with other objects
• May be composed of other objects
• Can be treated as a black box

Object-oriented programming (OOP) uses the concept of objects to model entities.

• A program consists of interactions between some number of objects
• Good object-oriented design involves making wise choices about which classes we create, and what each one represents

Object instances & classes

For our programs:

• We want to be able to use many objects (which may be similar to each other, or very distinct)
• We do not want to rewrite lots of code

In Java, we can achieve this by:

• Providing blueprints for all objects of the same type (group or kind) in a class, which specify

  Attributes or data members The properties of an object
  Behaviors or actions What an object can do

• Creating separate instances or objects of that class to use in our programs

Diagrams and implementations

A UML class diagram describes a class and how it can be used properly

• Sketch of attributes and behaviors for objects of that type
• No details about how it works — that’s in the Java implementation
public class Car {
    private String makeModel;
    private int mileage;
    public Car(String s, int m) {
        makeModel = s;
        mileage = m;
    }
    public void setMakeModel(String s) {
        makeModel = s;
    }
    public void setMileage(int m) {
        mileage = m;
    }
    public String getMakeModel() {
        return makeModel;
    }
    public int getMileage() {
        return mileage;
    }
}

- **Single responsibility principle** — a method should have one single and straightforward purpose

**Creating and using objects in a program**

**Creating an object**

```java
final Car myCivic = new Car("Honda Civic", 214118);
```

**Using an object**

```java
System.out.println(myCivic.getMakeModel());
myCivic.setMileage(1 + myCivic.getMileage());
```

- **Static** methods and fields are associated with the class as a whole
  - Do not need to create an object instance to use them
  - The main method of a class is the entry point for running that class as a program
  - These main methods should take a single String array argument, and declare a void return type

**Exercise 2.1.1.** Alongside the `Car` class, construct another class `CarLot` that tracks cars (such as for a used car dealership)

```java
public class CarLot {
    Car car1;
    Car car2;
    public CarLot() {
    }
    public void setCar1(Car c) {
        car1 = c;
    }
    public void setCar2(Car c) {
        car2 = c;
    }
    public void printCars() {
        System.out.println("Car 1: "+car1.getMakeModel());
        System.out.println("Car 2: "+car2.getMakeModel());
    }
    public Car getCar1() {
        return car1;
    }
    public Car getCar2() {
        return car2;
    }
}
```
The CarLot is an aggregate class, made up of other objects. In UML diagrams, the diamond arrow shows such a "used by" relationship as between Car and CarLot.

**Exercise 2.1.2.** Extend Exercise 2.1.1 by adding a `main` method to class CarLot which create one CarLot and two Car instances, populates the lot with the cars, and calls the `printCars` method of the lot.

**Exercise 2.1.3.** Write a class `Person` which implements the following UML:

```
Person
  - String name
  - int ticketsReceived

Constructor
  + Person(String)

Update
  + void noteTicketReceived()

Queries
  + String getName()
  + int getTicketsReceived()
```

where initially every person is assumed to have received zero tickets.

**Exercise 2.1.4.** Add a method `equals` to `Person` which

1. Takes a single argument of type `Person`, and
2. Requires only that the name fields of two `Person` instances are equal to return `true`.

---

**Where do objects live?**

We have already seen an example where objects can last longer than the stack frame of the method which created them

```java
public static Car buildMyCar() {
    final Car x = new Car("Honda Civic", 214118);
    return x;
}
```

and in some `caller`,

```java
final Car c = buildMyCar();
```

So to create an object, `new` allocates space in the heap

- The class name tells `new` how much space is needed, and how it will be used
  - Each `String` is also an object!
• The arguments are passed to the constructor to set up the new space

• The instance returned from `new` is assigned to `x`

• When `buildMyCar` exits, its result is assigned to `c`

**Static information is global**
Fields tagged `static` occur exactly once in memory

• Created at the beginning of program execution

• Remain until the end of program execution

```java
public class C {
    private static int x = 40;
    private int y = 50;
    // ...
}
```

And statements:

```java
C d1 = new C();
C d2 = new C();
```

Adding more complexity
Suppose the car dealership also sells commercial vehicles (trucks/vans) which have varying carrying capacities (e.g., 1/5/10 tons).
• Large amount of duplication
• Harder to write general-purpose code
  – New features in one class must be adapted to the other
  – Will need duplicate code to use Car and Truck instances
  – Harder to build a CarAndTruckLot where either type of vehicle can park in a slot

Finding a better solution
Certain types of objects have things in common

- Cars/trucks/motorcycles
- Savings/checking/investment accounts

We should adjust our model to exploit these commonalities.

- Via inheritance in object-oriented software design

Inheritance
Inheritance is when one class (the subclass or child class) is based on another class, the super class or parent class, which the child class extends or modifies in some way.

- Superclass (or parent) contains similarities
- Subclass (or child) extends the parent
  – Inherits methods and variables from the parent
  – Can add more methods and variables or modify existing ones

Allows us to make our code simpler and more useful!

Exercise 2.1.5. Consider your class Person of Exercise 2.1.3 and the description of class Student in Exercise 2.4.1. Rewrite this class Student so that it is a subclass of Person.
Inheritance in UML and Java

Inheritance can be represented in UML with arrows from children to parents

- Each child is a more specific kind of parent object
- Called an is-a relationship

public class Vehicle {
    // data and methods
}
public class Car extends Vehicle {
    // more data and methods
}
public class Truck extends Vehicle {
    // more data and methods
}

Everything is an Object

Objects can be part of an inheritance hierarchy, with multiple levels of ancestors and descendants.

- In Java, everything is descended from the Object class

Exercise 2.1.6. Write classes Vehicle, Car and Truck which implement the following UML:

You will need to use the superclass constructor super as the first step of the constructors for Car and Truck.
Exercise 2.1.7. Revise your Vehicle class from Exercise 2.1.6 to use the protected mode, indicated by the # tag in this revised UML design.

<table>
<thead>
<tr>
<th>Vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td># String makeModel</td>
</tr>
<tr>
<td># int mileage</td>
</tr>
</tbody>
</table>

**Constructor**

+ Vehicle(String, int)

**Update**

+ void setMakeModel(String)
+ void setMileage(int)

**Queries**

+ String getMakeModel()
+ int getMileage()

---

**Type conformance**

Every object *conforms* to the types of all its ancestors.

- In Java, everything conforms to the Object type

```
Diagram:

```

```

```

That is, in any place where a Vehicle instance is required, we could also provide either a Car or a Truck instance.

- Where an Object is required, we could provide an instance of Vehicle, or Car, or Truck — or any other class

---

**Conforming to a variable’s declared type**

One of the times when conformance comes up is when assigning to variables

- Whenever we declare a variable, we declare its type
- But we can assign values of other types into that storage location
  - So long as the type of the value conforms to the type of the variable
- In these cases, the variable’s declared type does not actually change

```java
Vehicle myVehicle;
myVehicle = new Car("Honda Civic", 214118);
myVehicle = new Truck("Ford F-150", 0, 2);
```

- For each assignment, the compiler makes sure that the type of the assigned object conforms to that of the variable
- If not, it is an error

```java
Truck myTruck;
myTruck = new Vehicle("home-built go-cart", 5); // Nope!
```
But here’s the tradeoff
   Conformance gives us flexibility in what we assign, but there is a cost

• When we declare a variable to have a certain type, we can use that variable only in ways which are sure to be valid for that type

• So this is fine:

    Vehicle myVehicle = new Car("Honda Civic", 214118);
    System.out.println(myVehicle.getMakeModel());

• But this is not:

    Vehicle z = new Truck("Ford F-150", 0, 2);
    System.out.println(z.getCapacity());

        – Even though class Truck does define the getCapacity method!
        – We judge the valid methods for z based on z’s declared type

No matter where conformance occurs, respect the declared bound
   This gap between the declared and actual types comes up in many places

With array entries
   Allowed

    Vehicle[]
        vehicles = new Vehicle[10];
        // ...
        vehicles[i] =
            new Car("Honda Civic", 214118);
    System.out.println(vehicles[i].getMakeModel());

   Forbidden

    Vehicle[]
        vehicles = new Vehicle[10];
        // ...
        vehicles[i] =
            new Truck("Ford F-150", 0, 2);
    System.out.println(vehicles[i].getCapacity());

With method parameters
   Allowed

• Method definition

    public static void m(Vehicle v) {
        System.out.println
            (v.getMakeModel());
    }

• Call

    m(new Car("Honda Civic", 214118));
Forbidden

- Method definition
  
  public static void m(Vehicle v) {
      System.out.println(v.getCapacity());
  }

- Call
  
  m(new Truck("Ford F-150", 0, 2));

Checking and casting

So we cannot just assume that some Vehicle is a Truck

But we can

- First explicitly check whether, in fact, it is one
- Then explicitly refer to the instance as a Truck

```java
final Vehicle vehicle = lot[i];
if (vehicle instanceof Truck) {
    final Truck truck = (Truck)vehicle;
    // ...use truck as a Truck here...
}
```

- The instanceof predicate tells us whether the type of an instance conforms to a class
- When we write (Truck)vehicle, we are downcasting

Casting does not duplicate the object instance

- It creates another reference to the same instance

---

Exercise 2.1.8. Write a static method `announceVehicleType` which

- Takes a Vehicle instance as an argument, and returns no value
- Prints exactly one of these three messages as appropriate:
  
  1. It’s a car!
  2. It’s a truck carrying ___ pounds, filling in the blank with the truck’s capacity
  3. I don’t know what that is!
Dispatching methods

Consider these classes

```java
class Parent {
    public void f() {
        System.out.println("A");
    }
}

class Child extends Parent {
    @Override
    public void f() {
        System.out.println("B");
    }
}
```

What do these lines print?

```java
final Parent x = new Child();
x.f();
```

Another question of declared type vs. actual type!

- Does Java follow to the declared type of the variable `x`?
- Or does Java follow the actual type of the object instance in the heap?

The rule in Java:

- It is the actual type of an object which determines how it behaves

Some vocabulary:

- We say that `f` is polymorphic
- The act of deciding what method corresponds to a reference to `f` is called dispatch
- The use of the actual type of the object to determine dispatch is called dynamic dispatch
  - Or virtual method invocation, if you come from the C++ world

Accessing overridden methods

When a class `B` overrides some method `f`, it can still invoke the version of `f` from its parent `A`

- In `B`'s methods, writing `super.f(...)` invokes the version of `f` defined for `A` instead of `B`
- Useful for extending a method or modifying its results

```java
public class Hatchback extends Car {
    // Sensible constructor not shown
    @Override public String toString() {
        return super.toString() + " with hatchback";
    }
}
```
• But be wary of overuse
  – Can lead to brittle and confusing code
  – Use of super across different methods can be a sign of poor design
  – Good usage is when a method calls its own overridden version — "do that old work, plus this here work too"

In these exercises work out for yourself, on paper, what the answers are before running the given code as Java to verify your work.

Exercise 2.1.9. Consider these classes:

```java
public class Obverse {
    protected int m;
    public Obverse(int mIn) {
        m = mIn;
    }
    public int getValue() {
        return m;
    }
}

public class Reverse extends Obverse {
    private int b;
    public Reverse(int a, int bIn) {
        super(a);
        b = bIn;
    }
    public int getB() {
        return b;
    }
    @Override public int getValue() {
        return m + b;
    }
}
```

After running the next six lines, what values will be assigned to `x`, `y` and `z`?

```java
Obverse a = new Obverse(1);
Reverse b = new Reverse(2,3);
Obverse c = b;

int x = a.getValue();
int y = b.getValue();
int z = c.getValue();
```

Exercise 2.1.10. Consider these classes:

```java
public class Curly {
    private final int c;
    public Curly(int cIn) {
        c = cIn;
    }
```
public int getCurliness() {
    return c;
}

public class Larry extends Curly {
    private final int l;
    public Larry(int c, int l) {
        super(c);
        l = lIn;
    }
    public int getLarrility() {
        return l;
    }
    @Override public int getCurliness() {
        return super.getCurliness() + getLarrility()/3;
    }
}

public class Moe extends Larry {
    private int m;
    public Moe(int c, int l, int mIn) {
        super(c, l);
        m = mIn;
    }
    @Override public int getLarrility() {
        return 6*m + super.getLarrility();
    }
}

What values will each of the following sequences of statements print?

1. Curly who = new Larry(3, 60);
   System.out.println(who.getCurliness());

2. Larry who = new Larry(3, 60);
   System.out.println(who.getCurliness());

3. Curly who = new Moe(3, 60, 900);
   System.out.println(who.getCurliness());

4. Larry who = new Moe(3, 60, 900);
   System.out.println(who.getCurliness());

Identity equality
In Java, we have two forms of equality for reference types: identity equality and content equality

Two objects have the same identity if and only if they are both the same object, at the same address in memory
if (obj1.equals(obj2)) {
    System.out.println("Same content");
}

Content equality

Two objects have the same content if and only if they have the same state (as defined by the programmer)

```
Person p1 = new Person("Joe Smith");
Person p2 = new Person("Joe Smith");
if (p1 == p2) {
    System.out.println("Same object");
}
if (p1.equals(p2)) {
    System.out.println("Same content");
}
```

Program output:

```
Same content
```

Defining content equality

By default content equality is defined as identity equality:

```
public class Object {
    //...
    public boolean equals(Object other) {
        return (this == other);
    }
}
```

To modify this behavior, we override the equals method
• But remember, equals is defined

    public boolean equals(Object other)

We can compare any two objects for equality

• Usually, we first check the type of the other object
  – If the types differ, we normally just return false right away
    
    if (!(other instanceof Vehicle)) {
      return false;
    }
  – If the types match, we can cast the argument

    final Vehicle v = (Vehicle)other;

Defining content equality — putting it all together

Once we cast the argument, we can access its fields, and reason about the objects’ equality

    public abstract class Vehicle {
      public boolean equals(final Object other) {
        if (!(other instanceof Vehicle)) {
          return false;
        }
        final Vehicle v = (Vehicle)other;
        return getMakeModel().equals(v.getMakeModel())
          && (getMileage() == v.getMileage());
      }
    }

Expectations for an equals method

The equals method should implement an equivalence relation on non-null object references:

• It should be reflexive:

    // Should always print true
    System.out.println(obj.equals(obj));

• It should be symmetric:

    // Should always print true
    if (objA.equals(objB)) {
      System.out.println(objB.equals(objA));
    } else {
      System.out.println(!objB.equals(objA));
    }

• It should be transitive

    // Should always print true, or nothing
    if (objA.equals(objB) && objB.equals(objC)) {
      System.out.println(objA.equals(objC));
    }

• There is no way for the compiler to enforce these expectations!
– But if we fail to meet these conditions, our programs can misbehave in mysterious (and very hard to debug!) ways
– More details on the Javadoc page for Object

See also Review Exercises D.4.31, D.4.33 and D.4.36

Exercise 2.1.11. A rational number is one which can be represented as a fraction of two integers. Define a class Rational for representing a rational number by storing the two integers of its fraction. Give it one construction of two int arguments, taking the numerator (the top number of the fraction) first. Write an equals method for your class which returns true when two instances represent the same rational number. For example, the instances

new Rational(1,2)
new Rational(3,6)

should be considered equal, but

new Rational(1,2)
new Rational(3,7)

should not.

Implementing the equals method

The implementation of equals provided in Vehicle compares Vehicle instances by make/model and mileage, and is an equivalence relation

• All subclasses (Car, Truck, Van) inherit this implementation.
• Given that, what is the output of the code below?

```
final Truck
    t1 = new Truck("Ford F-150", 0, 5),
    t2 = new Truck("Ford F-150", 0, 5);
System.out.println(t1.equals(t2));
t2.setCapacity(7);
System.out.println(t1.equals(t2));
```

– Both statements print true!
• Capacity of Truck instances is ignored
  – We never told Java that it should be considered
  – But easy to fix

The equals method for Truck instances

Truck instances should be compared by make/model, mileage and capacity

```java
public class Truck extends Vehicle {
    //...

    // Overrides equals method from Vehicle
    public boolean equals(Object other) {
        if (!(other instanceof Truck)) {
            return false;
        }
        Truck t = (Truck)other;
        return (getMakeModel().equals(t.getMakeModel())) &&
```

57
getCapacity() == t.getCapacity() &&
getMileage() == t.getMileage();

Avoiding duplication

Some of the comparisons are already encoded in the Vehicle.equals method

- We can avoid repeating those comparisons using super

```java
public class Truck extends Vehicle {
    //...

    public boolean equals(Object other) {
        if (!(other instanceof Truck)) {
            return false;
        }
        final Truck t = (Truck) other;
        return (this.capacity == t.capacity) && super.equals(other);
    }
}
```

Checking the implementation

With the equals method for Truck instances, this code behaves as expected:

```java
final Truck t1 = new Truck("Ford F-150", 0, 5),
t2 = new Truck("Ford F-150", 0, 5);
System.out.println(t1.equals(t2)); // true
System.out.println(t2.setCapacity(7));
System.out.println(t1.equals(t2)); // false
```

But what about this case?

```
final Truck t = new Truck("Ford F-150", 0, 5);
final Car c = new Car("Ford F-150", 0);
System.out.println(t.equals(c));   // [A]
System.out.println(c.equals(t));  // [B]
```

- Line [A] prints true, as we would hope
- But line [B] prints false!
- We have broken the symmetric property of equals
  - (Which means transitivity is also broken)

Implementing equals across subclasses

In general, it is very difficult to retain the symmetric and transitive properties for equals across subclasses.

- A solution: Have each subclass provide its own equals method

```java
public class Car extends Vehicle {
    //...

    public boolean equals(Object other) {
        //...
    }
}
```
if (!(other instanceof Car)) {
    return false;
}

Car c = (Car) other; // Cast Obj. ref to Car ref
return (getMakeModel().equals(c.getMakeModel()) &&
      getMileage() == c.getMileage());

• We run into the same issues if we have subclasses of Car, Truck, or Van.

2.1.1 Additional exercises for this section

Exercise 2.1.12. Recall the formula for compound interest from Exercise D.1.15:

\[ A = P \left(1 + \frac{r}{n}\right)^{nt}, \]

where:

- \( A \) is the total value at the end of the investment period,
- \( P \) is the original invested value,
- \( r \) is the interest rate as a decimal (so, 0.05 for 5%),
- \( n \) is the number of times per year that interest is compounded, and
- \( t \) is the (whole) number of years of investment.

In this exercise you will break up this formula into two stages, writing a class `InterestCalc` with both static and object methods.

Class `InterestCalc` should have:

- A constructor which takes a `double` representing the interest rate, and an `int` representing the number of compounding periods per year.
- An object method `getFinalValue` which takes two arguments,
  1. A `double` value representing the original invested value, and
  2. An `int` value representing the number of years invested, and
returns the final value of such an investment under the interest rate and compounding period associated with this object.
- A static method `getInterestCalculator` which takes the same two arguments as the constructor, and returns a new `InterestCalc` specialized to that rate and compounding period.

\[ \square \]

2.2 Abstract classes and methods

Writing method bodies

So far, when we have written classes

- We have always provided the method \textit{bodies}, the list of statements in curly braces
- Of course!
• Methods of objects might be called, and Java needs to know what steps to take

Contrast with object fields
The rules are different for object fields:
• We can declare them in one place in the class
• But set them only later, in the constructor or in a method

Is there any way to separate the declaration of a method signature from the definition of its body?
• And is there any sense to this?

Interfaces and abstract classes
In Java there are two ways to delay defining method bodies: interfaces and abstract classes

• We cannot instantiate either one of them with new
  – When an object is placed in the heap, its methods must all be defined
• To instantiate, we write other classes
  – Which implement the interface
  – Or extend the abstract class

Java can instantiate these other classes
  – So long as they are fully defined, and not themselves abstract

Interfaces
Interfaces specify methods without (normally) defining their bodies
• We provide only the signature
  – The names of the method
  – The number of parameters each method has
  – The type of each parameter
• Interfaces may not declare object fields, nor constructors
• May implement other interfaces
• May not extend classes

A class can implement one or more interfaces
• The class is then required to define a body for each interface method

Interfaces were designed to allow different implementations of the same method by different implementing classes
A vehicle interface

Every object conforms to both its ancestor classes and ancestor interfaces

```
public interface VInterf {
    public String getMakeModel();
    public String getMileage();
}
```

```
public class Vehicle
    implements VInterf {
    protected String makeModel;
    protected int mileage;
    // ...
}
```

**Exercise 2.2.1.** The following interface describes the signature of predicates over integers, that is, yes-or-no questions about integers.

```
public interface IntegerPredicate {
    public boolean check(int n);
}
```

Write these classes which implement `IntegerPredicate`:

1. Class `IntegerEvenPredicate`, whose `check` method returns `true` exactly when its argument is an even integer (where even means that the result when dividing by two has remainder 0).
2. Class `IntegerSquarePredicate`, whose `check` method returns `true` exactly when its argument is a perfect square.
3. Class `IntegerPowerOfTwoPredicate`, whose `check` method returns `true` exactly when its argument is an exact power of two (1, 2, 4, 8, 16, and so on).

**Exercise 2.2.2.** To apply Exercise 2.2.1, write a class `IntPredUser` with a static and `void` method `process` which

- Takes an argument of type `IntegerPredicate`
- Prints the result of passing 5 to the argument’s `check` method.

For example, the `main` method of `IntPredUser` might have the statements
process(new IntegerEvenPredicate());
process(new IntegerSquarePredicate());
process(new IntegerPowerOfTwoPredicate());

which would print true, false and false.

Exercise 2.2.3. The following interface describes the signature of single-argument functions over integers.

```
public interface IntegerFunction {
    public int apply(int n);
}
```

Write these classes which implement `IntegerFunction`:

1. Class `IntegerIncrementFunction`, whose `apply` method returns a value one greater than its argument.
2. Class `IntegerSquareFunction`, whose `apply` method returns the square of its argument.
3. Class `IntegerDirakFunction`, where:
   - The constructor of the class takes an argument `base`
   - The `apply` method on an instance returns 1 if the argument is the same as that instance’s `base`, and 0 otherwise.

Exercise 2.2.4. To apply Exercise 2.2.3 write a class `IntFnUser` with a static and `void` method `process` which

- Takes an argument of type `IntegerFunction`
- Prints the result of passing 20 to the argument’s `apply` method.

For example, the main method of `IntFnUser` might have the statements

```
process(new IntegerIncrementFunction());
process(new IntegerSquareFunction());
process(new IntegerDirakFunction(10));
process(new IntegerDirakFunction(20));
```

which would print 21, 400, 0 and 1.

Exercise 2.2.5. The following interface describes the signature of two-argument functions over integers.

```
public interface IntegerBinaryFunction {
    public int apply(int m, int n);
}
```

Write these classes which implement `IntegerBinaryFunction`:

1. Class `IntegerAddition`, whose `apply` method returns the sum of its arguments.
2. Class `IntegerMultiplication`, whose `apply` method returns the product of its arguments.
3. Class `IntegerTenTwist`, whose `apply` method returns the sum of the second argument, and 10 times the first argument.
Exercise 2.2.6. To apply Exercise 2.2.5, write a class `IntBinFnUser` with a static and void method `process` which

- Takes an argument of type `IntegerBinaryFunction`
- Prints the result of passing 5 and 10 to the argument's `apply` method.

For example, the `main` method of `IntBinFnUser` might have the statements

```java
process(new IntegerAddition());
process(new IntegerMultiplication());
process(new IntegerTenTwist());
```

which would print 15, 50 and 60.

Abstract classes

- Methods in an abstract class may be defined in full (as in normal classes)
- Methods may also be declared with a signature only (as in interfaces)
  - Tagged with the `abstract` keyword
- Abstract classes can have fields and/or constructors
- Abstract classes can extend other classes
  - Whether abstract or concrete
- Abstract classes can implement interfaces
  - Can give will definitions with bodies for none, some or all interface methods

Example: the car lot
Recall again our setup for `Vehicle`, `Car` and `Truck`

```
Vehicle
- String makeModel
- int mileage

Constructor
+ Vehicle(String, int)
Update
+ void setMakeModel(String)
+ void setMileage(int)
Queries
+ String getMakeModel()
+ int getMileage()
```

```
Car
Constructor
+ Car(String, int)
```

```
Truck
- int capacity

Constructor
+ Truck(String, int, int)
Update
+ void setCapacity(int)
Queries
+ int getCapacity()
```

- It would be odd to instantiate `Vehicle`
- We can make it abstract
Making **Vehicle** abstract

```java
public abstract class Vehicle {
    protected String makeModel;
    protected int mileage;

    public Vehicle(String s, int m) {
        this.makeModel = s;
        this.mileage = m;
    }
    public void setMakeModel(String s) {
        makeModel = s;
    }
    public void setMileage(int m) {
        mileage = m;
    }
    public String getMakeModel() {
        return makeModel;
    }
    public int getMileage() {
        return mileage;
    }
}
```

### Adding an abstract method

```java
public abstract class Vehicle {
    protected String makeModel;
    protected int mileage;

    public Vehicle(String s, int m) {
        this.makeModel = s;
        this.mileage = m;
    }
    public void setMakeModel(String s) {
        makeModel = s;
    }
    public void setMileage(int m) {
        mileage = m;
    }
    public String getMakeModel() {
        return makeModel;
    }
    public int getMileage() {
        return mileage;
    }
}
```
abstract Vehicle
- String makeModel
- int mileage

Constructor
+ Vehicle(String, int)

Update
+ void setMakeModel(String)
+ void setMileage(int)

Queries
+ String getMakeModel()
+ int getMileage()
+ abstract String getInfo()

The revised car lot

public class Car extends Vehicle {
    public Car(String s, int m) {
        super(s, m);
    }
    public String getInfo() {
        final StringBuilder info = new StringBuilder();
        info.append("This Car is a ");
        info.append(makeModel);
        info.append(" with ");
        info.append(mileage);
        info.append(" miles.");
        return info.toString();
    }
}

public class Truck extends Vehicle {
    private int capacity;
    public Truck(String s, int m, int c) {
        super(s, m);
        capacity = c;
    }
    public String getInfo() {
        info.append("This Truck is a ");
        info.append(makeModel);
        info.append(" with ");
        info.append(capacity);
        info.append(" miles, and a ");
        info.append(capacity);
    }

• Each child class extends the same abstract parent class

• Each child class provides its own implementation of the parent’s abstract methods

Using abstract classes

```java
public void fillLot() {
    final Vehicle[] vehicles = new Vehicle[10];
    for (int i=0; i<vehicles.length; ++i) {
        if (i % 2 == 0) {
            vehicles[i] = new Car("Honda Civic", 0);
        } else {
            vehicles[i] = new Truck("Ford F-150", 0, 10);
        }
    }

displayLot(vehicles);
}

private void displayLot(Vehicle[] vehicles) {
    for (int i = 0; i < vehicles.length; ++i) {
        System.out.println((i +": "+ vehicles[i].getInfo()));
    }
}
```

• The base type of the array is the abstract type
  – Stores references to objects that conform to Vehicle

• Array is filled with references to objects whose actual type is concrete
  – Of course: only concrete types can be instantiated!

• Calling `getInfo()` works for all objects
  – Based on the actual object type, Java dispatches the version appropriate for each

Roundup
<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Abstract classes</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can instantiate with <code>new</code></td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Can declare abstract methods</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can fully declare methods</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Can declare fields and constructors</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Can use with <code>extends</code></td>
<td>✗</td>
<td>One only</td>
</tr>
<tr>
<td>Can use with <code>implements</code></td>
<td>Zero or more</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Exercise 2.2.7.** Consider these two classes:

```java
public class Person {
    private String name;
    //...
    public String getInfo() {
        return "Name: " + name;
    }
}
```
```java
public class Animal {
    private String type;
    //...
    public String getInfo() {
        return "Type: " + type;
    }
}
```

Write an abstract class `Informative` which captures the signature of the `getInfo` methods, and rewrite the `Person` and `Animal` classes to extend this abstract class.

**Exercise 2.2.8.** Write a static method `showAllInfo` which

- Takes a single argument, an array of `Informative` instances,
- Returns no result,
- Prints the result of a call to `getInfo` to each element of the array, each on its own line

**Exercise 2.2.9.** Revise Exercise 2.2.7 so that `Informative` is an interface rather than an abstract class

**Exercise 2.2.10.** Consider the method `showAllInfo` from Exercise 2.2.8 using the definition of `Informative` from Exercise 2.2.9 instead. What, if anything, needs to change in `showAllInfo`?

67
Exercise 2.2.11. Consider the interface `Speaker`,

```java
public interface Speaker {
    public void speak();
}
```

Revise the classes `Person` and `Animal` of Exercise 2.2.9 so that both classes implement both `Informative` and `Speaker`. The result of running this code:

```java
final Speaker[] speakers = {
    new Person("Joan"),
    new Person("Betty"),
    new Animal("monster")
};
for(Speaker sp : speakers) {
    sp.speak();
}
```

should be:

- Hi! My name is Joan.
- Hi! My name is Betty.
- Woof! Woof!!

(We will simply pretend that all animals bark.)

So far you have learned *new language features* to solve new problems. But we are quickly approaching the point where we will have seen all of the major features of languages like Java. So our focus for solving harder problems will come to include:

- More sophisticated algorithms
- Understanding and using libraries
- Describing ideas which encompass several language structures and how they interact

Software design patterns address the last of these keys, and the next exercises consider two examples of design patterns. They give us a vocabulary for general, reusable solutions to commonly occurring problems within a given context in software design.

1. **Template Method** is a pattern for classes with related behavior. We apply Template Method when we define an outline (skeleton) of an algorithm in a template class, and provide implementation details in subclasses.

   For example, the top-level structure of a game could be defined as:

   ```java
   public abstract class Game {
       public void play() {
           initialize();
           while (!gameIsOver()) {
               takeATurn();
           }
       }
   }
   // The template methods
   protected abstract void initialize();
   protected abstract boolean gameIsOver();
   protected abstract void takeATurn();
   }
   ```

   Then particular game classes will provide the details for each step by defining versions of `initialize`, `gameIsOver` and `takeATurn` *without overriding* `play`. 
Exercise 2.2.12. The factorial method of Exercise D.2.17 and this sumThrough method have similar structure:

```java
public long sumThrough(int n) {
    long result = 0;
    for(int i=1; i<=n; i=i+1) {
        result = result + i;
    }
    return result;
}
```

(Assume for this exercise that the factorial method is not static). Write a class AbstractIntervalConsumingCalculation which can serve as a Template Method pattern for these two methods, using these abstract methods:

```java
public abstract class AbstractIntervalConsumingCalculation {
    public long performCalculation(int n) {
        // You fill this in
    }
    public abstract int getIntervalStart(int n);
    public abstract int getIntervalEnd(int n);
    public abstract long getInitialResult();
    public abstract long getUpdatedResult(long oldResult, int newIntervalMember);
}
```

Then write concrete classes DerivedFactorial and DerivedSumThrough which extend AbstractIntervalConsumingCalculation, where the performCalculation method on instances of each returns the same results as the original factorial and sumThrough methods. Your DerivedFactorial and DerivedSumThrough classes must not override the performCalculation method! They should only override the abstract methods with full definitions of those methods.

Factory Method is a creational pattern where instead of using constructors directly within code, we define an interface for object creation. This interface has factory methods which returns a new instance. For example, this interface VehicleFactory expresses the general action of creating a Vehicle without committing to a particular form (concrete subclass) of Vehicle:

```java
public interface VehicleFactory {
    public Vehicle build(int mileage);
}
```

Then we can define different factory object for instantiable subclasses of Vehicle:

```java
public class VolvoFactory extends VehicleFactory {
    public Vehicle build(final int mileage) {
        return new Car("Volvo", mileage);
    }
}
public class FordTruckFactory extends VehicleFactory {
    public Vehicle build(final int mileage) {
        return new Truck("Ford F-650", mileage, 3);
    }
}
```

Note that factory methods may take arguments, in just the same way as constructors.
Exercise 2.2.13. Consider your class `Person` of Exercise 2.1.3. Write a class `PersonFactory` which implements the following interface to establish a Factory Method pattern:

```java
public interface PersonFactoryMethod {
    public Person getPerson(String firstName, String lastName);
}
```

Exercise 2.2.14. Write a class `StoogesForever` with a single static method with signature

```java
public static Person[] makeStoogesArray(PersonFactoryMethod factory)
```

Your `makeStoogesArray` method should create an array of size three, and populate with three `Person`-conforming objects derived from its `factory` argument, the three objects corresponding to Moe Howard, Larry Fine, and Shemp Howard. Test your class and method by providing it an instance of your `PersonFactory` from Exercise 2.2.13.

Exercise 2.2.15. Students are people too! (Allegedly) Use your class `Student` from Exercise 2.1.5 to write a second class `StudentFactory` which also implements the `PersonFactoryMethod` interface, but where the result of `getPerson` has a runtime type of `Student` instead of `Person`. Your `makeStoogesArray` method of Exercise 2.2.14 should be just as happy with an instance of `StudentFactory` as it is with an instance of `PersonFactory`.

The next two exercises use the `java.util.Random` class, which is part of the standard Java libraries and is documented on the API Javadoc pages, to generate random numbers.

Exercise 2.2.16. Write a class `RandomStringFactory` whose `build()` method produces a string whose length is between 0 and 255, containing randomly selected printable ASCII characters.

Exercise 2.2.17. Write a class `RandomNewVehicleFactory` whose `build()` method produces, at random, either a `Car`, `Truck` or `Hatchback` instance, with

- The make and model chosen from a small number of plausible strings, and
- Zero mileage.

## Checking the actual class

Abstract classes can give us another technique for defining **equal** across subclasses

- The class of an object is available at runtime
- Sometimes, this can help us avoid problems when comparing across subclass
- **Object** also defines the method `getClass`
  - Returns the actual, runtime type of the object
  - Expressed as an object of type `java.lang.Class`
- So in `Vehicle` we can check the actual classes of this and other:

---

1 Introductory information about design patterns is available at [en.wikipedia.org/wiki/Software_design_pattern](en.wikipedia.org/wiki/Software_design_pattern) Gamma et al.’s book [1] was a breakthrough publication which introduced design patterns to a much wider audience, and it remains a valuable reference.
public abstract class Vehicle {
    //...

    public boolean equals(Object other) {
        if (getClass() != other.getClass()) {
            return false;
        }
        final Vehicle v = (Vehicle)other;
        return getMakeModel().equals(v.getMakeModel())
               && (getMileage() == v.getMileage());
    }
}

2.3 More class forms

Classes and files
In older versions of Java, each class lived in its own file

• This is still substantially the case

But early on there were no exceptions!

• So programs tended to have many files

• Too many files

So Java evolved to have ways to specify classes without creating new files

• Inner classes
  – Static
  – Object

• Anonymous inner classes

• Lambda expressions

Static inner classes
An inner class is a class defined inside the scope of another class

• Like the fields and methods of the enclosing class, inner classes can be referenced anywhere within the enclosing class

• If the inner class is declared public, then it can be accessed from outside the enclosing class as well

So given

```
public class Enclosing {
    // ...fields and methods...

    public static class Inside {
        public Inside(int a) { // ...
    }
}
```

we could instantiate new Enclosing.Inside(10).

• There is a single definition of Inside
Inner classes with an enclosing object

When an inner class is non-static, it can refer to the fields of the object which encloses it.

```java
public class Enclosing {
    private int i;
    private Inside hidden;
    public Enclosing (int i) {
        this.i = i;
        this.hidden = new Inside();
    }

    public void unhide() {
        hidden.peek();
    }
}

public class Inside {
    public void peek() {
        System.out.println(i);
    }
}
```

- **Note that** \( i \) **is used within** Inside, **but is not defined within the class**
- **The Enclosing constructor creates an Inside instance specific to its own value of** \( i \)
- **If** Enclosing **has this main routine**

```java
public static void main(String[] args) {
    Enclosing e1 = new Enclosing(1);
    Enclosing e2 = new Enclosing(2);
    e1.unhide();
    e2.unhide();
}
```

it would print 1 and 2.

Understanding the inner classes

One way to understand inner classes is to imagine a separate class with an additional argument for the parent:

With an inner class

```java
public class Enclosing {
    private int i;
    private Inside hidden;
    public Enclosing (int i) {
        this.i = i;
        this.hidden = new Inside();
    }

    public void unhide() {
        hidden.peek();
    }
}

public class Inside {
    public void peek() {
    }
```
With separate classes

```java
public class Enclosing {
    int i;
    private Inside hidden;
    public Enclosing (int i) {
        this.i = i;
        this.hidden = new Inside(this);
    }

    public void unhide() {
        hidden.peek();
    }
}
```

```java
public class Inside {
    private Enclosing parent;
    Inside(Enclosing parent) {
        this.parent = parent;
    }
    public void peek() {
        System.out.println(parent.i);
    }
}
```

But inner classes are more powerful than just this "trick"

- Inner classes work even when `i` is declared `private`!

**Exercise 2.3.1.** Consider this interface

```java
public interface SteeringWheel {
    public Vehicle controlsVehicle();
    public boolean hasRadioControls();
    public boolean hasClimateControls();
}
```

First, extend the `Vehicle` class with a method

```java
public SteeringWheel getSteeringWheelInfo();
```

Then extend each subclass of `Vehicle` to include an inner class definition which is instantiated by the `Vehicle` subclass's constructor in order to return a result from `getSteeringWheelInfo`.

**Anonymity**

With inner classes, we always give the class a name

- Of course!
- It's part of a class declaration
- It's how we use the class

But names are not always convenient, especially if we must name everything

- This assignment is easy to understand:

  \[ y = (x + 4) \times (z - 4) \times (x + z); \]

- If we must name every intermediate result, it obfuscates the point of the code:

  \[
  \begin{align*}
  &\text{int } xp4 = x + 4; \\
  &\text{int } zm4 = z - 4; \\
  &\text{int } xpz = x + z; \\
  &\text{int } \text{prod1} = xp4 \times zm4; \\
  &\text{int } y = \text{prod1} \times xpz; \\
  \end{align*}
  \]

- It can become a burden to think of so many consistent, distinct, mnemonic names

Java lets us write anonymous inner classes

- For situations where we will instantiate the class in one place only
- We will not even refer to the class elsewhere, so it does not actually need a name

How to write an anonymous class

- An anonymous class must have exactly one base type
  - Could be just Object
  - Could be an interface, an abstract class, or a concrete class
- Since we are immediately instantiating the anonymous class, if its base is a class there may also be constructor arguments
- Since the base could be abstract, an anonymous class has a body of method definitions
  - Can override other methods as usual

```java
Vehicle v = new Car("Bentley Mulsanne", 10) {
    @Override public String toString() {
        return "A discrete and bespoke sedan";
    }
};
```

**Exercise 2.3.2.** In the sample main method of Exercise 2.2.2, replace the three given object instantiations with anonymous inner class instantiations based on `IntegerPredicate`. 

**Exercise 2.3.3.** In the sample main method of Exercise 2.2.4, replace the three given object instantiations with anonymous inner class instantiations based on `IntegerFunction`. 

**Exercise 2.3.4.** In the sample main method of Exercise 2.2.6, replace the three given object instantiations with anonymous inner class instantiations based on `IntegerBinaryFunction`. 

74
Are anonymous classes a bit wordy?

There is a special case where Java will let us omit some of the boilerplate from an anonymous inner class instantiation

- When we are providing an anonymous inner class instantiation for an interface with a single method
- When the context of that instance is unambiguous as to what interface is required

Then we can use a lambda expression to abbreviate the anonymous class

**How to write a lambda expression**

- Exercise [2.2.5](#) presented the interface

  ```java
  public interface IntegerBinaryFunction {
    public int apply(int m, int n);
  }
  ```

  and asked us to write classes like

  ```java
  public class IntegerAddition
      implements IntegerBinaryFunction {
    @Override public int
        apply(int m, int n) {
          return m+n;
        }
  }
  ```

- Exercise [2.2.6](#) discussed a method `process` taking an `IntegerBinaryFunction` instance as an argument, where we might expect a call

  ```java
  process(new IntegerAddition());
  ```

- With anonymous classes, we could already dispense with an explicit tiny class like `IntegerAddition`:

  ```java
  process(new IntegerBinaryFunction() {
    @Override public int
        apply(int m, int n) {
          return m+n;
        }
  });
  ```

  - This call meets the requirements for a lambda expression, so we can simplify to:

    ```java
    process((int m, int n) -> m+n);
    ```

  - In fact, because Java can see that only `int` parameters are allowed for `apply`, we can simplify further to:

    ```java
    process((m, n) -> m+n);
    ```

    Concise!

---

**Exercise 2.3.5.** Simplify the calls from Exercise [2.3.2](#) to use lambda expressions.

**Exercise 2.3.6.** Simplify the calls from Exercise [2.3.3](#) to use lambda expressions.

**Exercise 2.3.7.** Simplify the other calls from Exercise [2.3.4](#) to use lambda expressions.
3 Arrays

3.1 Arrays under the hood

What happens when we create an array?

```java
final ElementType[] array = new ElementType[arraySize];
```

An array can last longer than the stack frame of the method where it is allocated.

```java
public static ElementType[] makeArray(int arraySize) {
    final ElementType[] a = new ElementType[arraySize];
    return a;
}
```

So arrays are allocated in the heap.

![Array Allocation Diagram]

What about the contents?

The location of the array’s contents depends on whether the array holds elements of *primitive* type or of *reference* type.

Values of primitive types are stored in the array itself.

![Primitive Type Contents]

Values of reference type have their own heap space.

![Reference Type Contents]

Arrays are always initialized

Java’s security model requires that all array slots be given initial values.

For primitive types, this is zero or false.

```java
public static int[] makeArray(int arraySize) {
    final int[] a = new int[arraySize];
    return a;
}
```
For reference types, this is the null pointer, literally pointing to nothing

```
public static MyClass makeArray(int arraySize) {
    final MyClass[] a = new MyClass[arraySize];
    return a;
}
```

Arrays as parameters
Since an array is a heap object, the local value associated with it is the reference to that heap location

- This means that when we pass an array as a parameter, we pass the reference
- Not the contents!

```
public static void main(String[] args) {
    int[] a = { 1, 2, 3 }
    squareArray(myArr);
}
```

```
public static void squareArray(int[] arr) {
    // ... 
}
```

When an array is passed to a method, only its reference is passed (just like objects)
- Any modifications that the method does to the array contents persist after the method ends

Array arguments and mutation
Since the reference is shared, any changes made to the array will be shared, and will persist

```
public static void main(String[] args) {
    int[] a = { 1, 2, 3 }
    squareArray(myArr);
}
```

```
public static void squareArray(int[] arr) {
    // ... 
}
```
Exercise 3.1.1. Trace the stack and heap through the execution of this program:

```java
public static void modifyArray(int[] arr) {
    arr[0] = 7;
    arr = new int[3];
    arr[2] = 9;
}
```

```java
public static void main(String[] args) {
    int[] myArr = {1, 2, 3};
    modifyArray(myArr);
    System.out.println(myArr[2]);
}
```

3.2 Search and its cost

Searching for information

- Finding a book in a library or name in an address book
- Finding movie show times & nearby locations
- Finding a path through a maze
- Finding the shortest drive from La Crosse to Las Vegas
- Finding a flight from La Crosse to London costing less than $1,200
Simple searching
Some of these types of searches are challenging, some are easier.

- Depends on the constraints of the search and the structure of the search space

An often critical factor in search is how our data is organized:

- Which data structures are we using?
- How can we access individual pieces of data?

A data structure is a particular way of organizing data in a computer (program) so that it can be used efficiently.

Example: How can be find a single piece of data in an array?

Linear search
Consider how to explain the search process step-by-step in English:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3-1</th>
<th>1-2</th>
<th>1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>78</td>
<td>81</td>
</tr>
</tbody>
</table>

1. Start at the beginning of the array
2. Check if the cell contains what you are looking for
3a. If it does, then report success and stop
3b. Otherwise, move on to the next cell and repeat, assuming you aren’t at the end of the array
4. If you get to the end of the array and haven’t found the item, report failure

Implementing linear search
Take the English description and isolate the basic parts

1. Start at the beginning of the array
2. Check if the cell contains what you are looking for
3. If it does, then report success and stop
   - Otherwise, move on to the next cell and repeat, assuming you aren’t at the end of the array
4. If you get to the end of the array and haven’t found the item, report failure
Linear search in Java

After identifying the basic parts, translate into code:

```java
private static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; ++i) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Why return -1?

- If the search succeeds, the method returns the position of the target item within the array.
- If the search fails, the method returns a signal value of -1 to indicate to the caller that the target has not been found.
  - -1 is definitely not an index of the array!

Exploiting the structure of the data

Can we do better than linear search?

- If the input array is a random list of numbers, then probably not
  - The target number could be anywhere!
  - Example: Finding a particular word in a book: It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled . . .

- If the input array is organized in some way, then maybe!
  - Example: Finding a particular word in a dictionary

```
dystopia: 1. an imaginary place where people lead dehumanized and fearful lives
```

https://www.merriam-webster.com/dictionary/dystopia

Binary search

We can take advantage of sorted data to improve the search process.
Binary search in Java

How might we implement binary search in Java?

```java
private static int binarySearch(int[] arr, int target) {
    int begin = 0;
    int end = arr.length - 1;

    while (begin <= end) {
        int mid = (begin + end) / 2; // Find the midpoint
        if (arr[mid] == target) { // Found it!
            return mid;
        } else if (arr[mid] < target) { // mid value too small
            begin = mid + 1;
        } else {/* arr[mid] > target */ // mid value too large
            end = mid - 1;
        }
    }
    return -1; // Failed search
}
```

Binary search: a recursive implementation

```java
private static int binarySearch(int[] arr, int target) {
    return binSearchHelper(arr, target, 0, arr.length - 1);
}

private static int binSearchHelper(int[] arr, int target, int begin, int end) {
    if (begin > end) { // Base case #1
        return -1; // Failed search
    }
    int mid = (begin + end) / 2; // Find the midpoint
    if (arr[mid] == target) { // Found it! (base case #2)
        return mid;
    } else if (arr[mid] < target) { // mid value too small
        return binSearchHelper(arr, target, mid + 1, end);
    } else {/* arr[mid] > target */ // mid value too large
        return -1; // Failed search
    }
}
```
Comparing linear and binary search

Binary search is more complicated than linear search — is this complexity worth it? How would we assess this?

- For both search algorithms, the worst case we could have for running time is when the item being searched for is not in the array.
- For linear search, the running time is $O(n)$.
  - Looking at an element requires a constant amount of work
  - Need to look at each element in the array
- For binary search, the running time is $O(\log n)$.
  - Finding the midpoint and inspecting the element requires a constant amount of work
  - Need to halve the array about $\log_2 n$ times

Is the difference meaningful?

Consider searching through an array of $2^{48}$ integers

- Each integer is 4 bytes, so total space is $2^{50}$ bytes, or one petabyte
- Around 2009, Google was processing 24 petabytes per day

Suppose it takes a nanosecond ($10^{-9}$ seconds) to process each entry

- Linear search:
  \[
  2^{48} \text{ entries} \times \frac{10^{-9} \text{ seconds}}{\text{entry}} \approx 2.81 \times 10^5 \text{ seconds} \approx 3.25 \text{ days}
  \]
- Binary search:
  \[
  \log_2(2^{48}) \text{ entries} \times \frac{1 \text{ ns}}{\text{entry}} = 48 \text{ ns} = 48 \text{ billionths of a second}
  \]

The difference between $O(n)$ and $O(\log n)$ can be dramatic!

3.3 Sorting

Sorting an array

To use binary search, the array must to be sorted

![Sorted Array]

There are many ways to do this.

But first, how can we tell if an array is sorted?
Identifying an unsorted array

public static boolean isSorted(int[] array) {
    // TODO: Implement this!
}

Bubble sort: overview

- Apply the scan-and-swap strategy to the array below:

```
2 1 3 6 0
```

- Compare the first pair of elements:

```
2 1 3 6 0
```

- Because the second element is smaller than the first, swap them.

```
1 2 3 6 0
```

Bubble sort: overview

- Repeat this process for subsequent pairs of elements:

```
1 2 3 6 0
```

```
1 2 3 0 6
```

- One scan through the array is not sufficient!

- We have moved the largest element to the rightmost slot
  - But after one pass, that’s all we can be sure of
Bubble sort: logical structure

1. Start at the beginning of the array
2. Check if first two elements are ordered correctly; if not, swap them
3. Repeat the process for subsequent pairs of elements
4. If no swaps were made, stop: the array is sorted
   • Otherwise, return to step 1 and repeat

Bubble sort: Java implementation

```java
public static void bubbleSort(int[] array) {
    boolean swapped;
    int numPasses = 0;
    do {
        swapped = false;
        for (int i=0; i<array.length-1-numPasses; ++i) {
            if (array[i] > array[i+1]) {
                int temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                swapped = true;
            }
        }
        ++numPasses;
    } while (swapped);
}
```

After each scan, the largest remaining element gets moved to the correct position, allowing us to stop the inner loop earlier each time

Bubble sort: complexity

The work done by `bubbleSort` is determined by how many times each loop executes

• Each pass through places the largest remaining item into its correct position, so at most \( n - 1 \) passes are required
• On the \( i^{th} \) pass, we have to look at \( n - i \) pairs of elements
• Looking at a pair of elements and swapping them if needed requires a constant amount of work

\[
\text{Total Work} = \sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \ldots + (n-(n-1)) = \sum_{i=1}^{n-1} i = O(n^2)
\]

Bubble sort: the best and worst cases

In the best case, the array is already sorted

- `bubbleSort` requires one pass through the array to verify that no swaps are necessary: \( O(n) \)
In the worst case, the array is sorted in the reverse order

5 4 3 2 1

- First pass moves the largest element to the end but leaves the remaining elements in the same relative ordering — $O(n^2)$ work in total

4 3 2 1 5

Selection sort: overview

Instead of scanning and swapping when we find an incorrect ordering, we could scan to find the smallest element, then move it to the beginning.

- Scan to find the smallest entry:

2 6 3 1 0

- Swap it into place:

0 6 3 1 2

- Repeat scan to find the next smallest entry and swap it into place:

0 1 3 6 2

Selection sort: logical structure

1. Start with the entire array marked "unsorted"
2. Scan through the unsorted portion to find the smallest element
3. Swap the smallest element with the element at the start of the unsorted portion; shrink the unsorted portion by one position
4. Repeat the process until there is no more unsorted portion

Selection sort: Java implementation

```java
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length; ++i) {
        int indexOfMin = i;
        for (int j = i + 1; j < array.length; ++j)
            if (array[j] < array[indexOfMin]) {
                indexOfMin = j;
            }
        if (i != indexOfMin) {
```
int temp = array[i];
array[i] = array[indexOfMin];
array[indexOfMin] = temp;
}
}
}

Selection sort: complexity
The work done by selectionSort is determined by how many times each of the loops executes.
• We have $n$ iterations of the outer loop
• On the $i^{th}$ iteration, the inner loop executes $n-i$ times
• Comparing two elements requires a constant amount of work
• Swapping a pair of elements requires a constant amount of work

Worst-case running time is then $O(n^2)$
• Run time is the same regardless of whether or not the input is already sorted
• Same worst-case performance as Bubble Sort, but fewer swaps

Merge sort: overview
Let’s try to apply the same idea we used for binary search to get better performance:
• A divide and conquer algorithm works by repeatedly breaking down a problem into smaller and smaller subproblems, until those subproblems become easy enough to be solved directly. The solutions to the subproblems then get pieced back together to provide a solution to the original problem.

• A sorting problem can be decomposed into smaller sorting problems
• Sorting a single element is an easy problem (base case)
• Subproblems can be recombined by merging their solutions together

Merge sort: overview

86
Merge sort: merging two lists

• To merge two lists, start at the beginning of each one:

    2 5 6 8  
  0 3 4 9

• Take the smaller element and place it in the new list:

    2 5 6 8  
  0 3 4 9

• Advance the position counters:

    2 5 6 8  
  3 4 9

• Take the smaller element and place it in the new list:

    2 5 6 8  
  3 4 9

• Repeat this process:

    5 6 8  
  3 4 9

• Until we get to the end:

    0 2 3 4 5 6 8 9
**Merge sort: Java implementation**

The most intuitive way to implement merge sort is with recursion!

```java
public static int[] mergeSort(int[] array) {
    if (array.length > 1) { // Check stopping condition
        int mid = array.length / 2 - 1;

        // Split array contents into two smaller arrays
        int[] left = copyRange(array, 0, mid);
        int[] right = copyRange(array, mid+1, array.length-1);

        // Recursively sort the smaller arrays
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves back together
        return merge(left, right);
    }
}
```

**Merge sort: complexity**

Given an array with \( n \) elements:

merge does not need just time...

But

**Merge sort: space complexity**

Merge sort does \( O(n \log n) \) total work

- But it also allocates \( O(n \log n) \) total space
- The other algorithms sorted in place
- We can simplify pretty easily to \( O(n) \) space — one spare buffer, and merge back-and-forth with the original space
- **But can we do better?**

**Quicksort**

- Also called partition-exchange sort
- Invented by Tony Hoare in 1959
Refined over the years

Quick sort is the default sorting algorithm in Java’s standard libraries

But it was revised as recently as 2009 in Java 7

Quick sort: basic idea

1. Choose one element of the sequence, the *pivot*

2. Rearrange elements of the list so that:
   - Everything less than the pivot is to the left of the pivot
   - Everything greater than the pivot is to the right of the pivot
   - (Does not really matter what we do with equal values)

3. Recur on the values to the left and right of the pivot

- Sorts the array in place
- Choose?
  - The performance of quicksort depends crucially on the choice of the pivot
  - We’ll come back to this point later

Quick sort structure

```java
public static void quicksort(int[] array, int lo, int hi) {
    if (lo<hi) {
        final int p = partition(array,lo,hi);
        quicksort(array, lo, p-1);
        quicksort(array, p+1, hi);
    }
}
```

Delegate to partition
- Choosing the pivot
- Rearranging the array elements about the pivot
- Returning the index of the pivot

Quick sort partitioning
- For the pivot, choose element \( hi \)
- Loop maintains indices \( i \) and \( j \):
  - The pivot is bigger than entries from \( lo \) to \( i \) (inclusive)
    1. Initially we have found no such entries
    2. \( i \) starts off as \( lo-1 \)
  - Entries from \( i+1 \) to \( j-1 \) are bigger than the pivot
    1. Initially we have found no such entries
    2. \( j \) starts off as \( lo \)
Entries from \( j \) to \( hi-1 \) are to be arranged

1. The loop places entry \( j \)

- After the loop, we swap the pivot to between these regions

- Check to see if needed

```java
public static int partition(int[] array, int lo, int hi) {
    int i = lo-1;
    for(int j=lo; j<hi; j++) {
        if (array[j]<pivot) {
            i += 1;
            int tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
        }
    }
    final int pivotPoint = i+1;
    if (pivot < array[pivotPoint]) {
        array[hi] = array[pivotPoint];
        array[pivotPoint] = pivot;
    }
    return pivotPoint;
}
```

3.3.1 Quicksort

**How does Quicksort perform?**

The `for` loop of `partition` visits every element of the (sub)list

- As with merge sort, the important question is how many times we do that

Some days, we are lucky

- If the pivot is near the middle of the range of values, we divide what we’re sorting about in half
- Then the analysis is as for merge sort: \( O(n \log n) \)

Some days, we are unlucky

- If the pivot is the highest or lowest value, we decrease the size of the unsorted area by one
- Then the analysis is as for selection sort: \( O(n^2) \)

Will we be lucky?

**Quicksort: the average case**

The worst case of QuickSort is that we are unlucky

- But in practice, this case is quite rare

QuickSort can be shown to have an *average* performance which really is \( O(n \log n) \)
• We can also push QuickSort towards $O(n \log n)$ performance by working harder on choosing the pivot
• Idea: take a constant amount of time to choose the pivot
• Or sometimes: take a non-constant time to choose the pivot for a greater average performance increase
• The current Java implementation
  – Uses an $O(n^2)$ for small arrays (below about 20)
  – Otherwise use a version of QuickSort with two pivots
  – Consistently runs faster in the average case than traditional QuickSorts — and Sun tested heavily before switching their implementation!

3.4 Two-dimensional arrays

Multi-Dimensional Arrays
In Java, arrays can be extended to more than one dimension.

• A one-dimensional array:

```java
int[] arr1d = new int[6];
arr1d[3] = 7;
```

• A two-dimensional array:

```java
int[][] arr2d = new int[3][5];
arr2d[1][2] = 4;
```

• Accessing dimensions:

```java
int[][] matrix = new int[7][10];
int numRows = matrix.length; // Returns 7
int numCols = matrix[0].length; // Returns 10
```

Using multi-dimensional arrays
Multi-dimensional arrays are useful for storing data that has multiple indices
• That is, "keys" to look it up
For example, storing movie reviews across users
final int numPeople = 3;
final int numMovies = 5;
final int[][] ratings =
    new int[numPeople][numMovies];

// ...

ratings[0][3] = 5;

<table>
<thead>
<tr>
<th>movie (2nd index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
</tr>
<tr>
<td>[0]</td>
</tr>
<tr>
<td>[1]</td>
</tr>
<tr>
<td>[2]</td>
</tr>
</tbody>
</table>

Multi-dimensional arrays
...and on to higher dimensions

- A one-dimensional array
  int[] arr1d = new int[6];
  arr1d[3] = 7;

- A two-dimensional array
  int[][] arr2d = new int[3][5];
  arr2d[1][2] = 4;

- A three-dimensional array
  int[][][] arr3d = new int[2][2][4];
  arr3d[0][1][2] = 6;

First index is like the page number of a notebook

• And so on
Using multi-dimensional arrays
Another example: hourly temperatures for a weather station over 3 years

```java
int years = 3;
int days = 365;
int hours = 24;
double[][][] temps =
    new double[years][days][hours];
```

Storing temperature of $-1.2$ for Year 2 of 3, January 01, at 12 noon:

```java
temps[1][0][12] = -1.2;
```

Using multi-dimensional arrays
Just as a single for loop is useful for manipulating a one-dimensional array, nested for loops are useful for manipulating an n-dimensional array

- One loop per dimension
  ```java
  int[][] arr2d = new int[20][15];

  for(int row=0; row<20; ++row) {
      for(int col=0; col<15; ++col) {
          final int n = arr2d[row][col];
          System.out.print(n + " ");
      }
  System.out.println();
  }
  ```

- row loops over the first dimension
- col loops over the second dimension

- This code does work, but what is wrong with it?

Avoid "magic numbers" in code
Hard-coding values leads to fragile code — difficult to maintain, hard to debug

Fragile!

```java
int[][] arr2d = new int[20][15];
for (int row=0; row<20; ++row) {
    for (int col=0; col<15; ++col) {
        int n = arr2d[row][col];
        System.out.print(n + " ");
    }
  System.out.println();
}
```

Solid

```java
int[][] arr2d = new int[20][15];
for (int row = 0;
    row < arr2d.length;
    ++row) {
    for (int col = 0;
        col < arr2d[row].length;
```
++col) {
    int n = arr2d[row][col];
    System.out.print(n + " ");
} System.out.println();

Use length to find both
  • The number of rows, and
  • The number of columns in each row

Arrays of arrays
A two-dimensional array is actually an array of arrays!

// Allocate space for 10 references to int[]
final int[][] arr2d = new int[10][];

// Allocate space for each "row"
for(int i=0; i<arr2d.length; ++i) {
    arr2d[i] = new int[5];
}

• arr2d is a variable that contains a reference to an array
  – arr2d.length gives size of this array
  – arr2d[i] gives element at position i

• arr2d[i] stores a reference to another array
  – arr2d[i].length gives size of this other array
  – arr2d[i][j] gives element at position j in this other array

Ragged arrays
What happens if we make this change to the array builder?

// Allocate space for 10 references to int[]
final int[][] arr2d = new int[10][];

for(int i=0; i<arr2d.length; ++i) {
    // Allocate space for each "row"
    arr2d[i] = new int[i+1];
}

Creates a ragged array (as opposed to a rectangular array)
Exercise 3.4.1. Draw the memory allocated for this array:

```java
// Allocate all space for array at once
int[][] matrix = new int[5][4]; // 5 rows and 4 columns

// Shortcut initialization: 2d array with 2 rows and
// 3 columns
int[][] matrix = {{3, 5, 6}, {2, 4, 7}};
```

Exercise 3.4.2. Draw the memory allocated for this array:

```java
// Allocate memory for each row separately
int[][] matrix = new int[3][]; // 3 rows
matrix[0] = new int[5]; // 5 columns in row 0
matrix[1] = new int[3]; // 3 columns in row 1
matrix[2] = new int[7]; // 7 columns in row 2

// Shortcut init.: 2 rows with 2 and 4 cols, respectively
int[][] matrix = {{2, 4}, {7, 3, 5, 6}};
```

Exercise 3.4.3. Assuming that the matrix in the skeleton below is initialized, write the code necessary to multiply every entry by scalar.

```java
double[][] matrix;
// Assume matrix initialized here
double scalar = ...;

// Your code here...
```

Exercise 3.4.4. Write a static method `matrixContains` that takes a 2-dimensional array of integers and an integer and returns `true` if the matrix contains that value, `false` otherwise.
4 Data structures

Self-referencing classes

We have used to classes that are composed of primitives plus other, simpler types of data objects.

```java
public class Car {
    private int mileage;
    private String makeModel;
    // ...and so on...
}
```

Can a class contain fields of the same class?

```java
public class MyClass {
    private int num;
    private MyClass c;
    // ...
}
```

Self-referencing classes

When we write a class, we are defining a sort of template

- Referencing other instances of the same class within the template is perfectly fine
- No object instances are created when the template is specified.
- The template specification simply declares variables that can be used to reference such objects — there is no immediate (and thus "infinite") recursion

A self-referencing class

```java
public class NumberNode {
    private int num;
    private NumberNode next;
    // ...
}
```

A self-referencing chart

![Percentage of Chart Which Resembles Pac-man](chart.png)
4.1 Linked lists

Linked data
Self-referencing classes allow us to link multiple instances of a class together

- The resulting linked allocations of memory are called data structures

There are many, many interesting data structures, but the simplest interesting structure is the linked list

A linear collection of nodes, each with a data value and a pointer to the next node in the list

Exercise 4.1.1. Define a Java class NumberNode for housing the fields num and next illustrated above.

- It should have three constructors, one taking values for both num and next; one taking only an initial value for num, and setting next to null; and the other taking no arguments and using 0 and null.
- The fields should be private, with setters and getter getNum etc. for both fields.

Exercise 4.1.2. Using your class NumberNode from Exercise 4.1.1 write Java statements which create this list:

Diagram the state of the stack and heap after each step of your code.

Linked lists are flexible
A linked list can be made arbitrarily long, simply by adding on more objects

- At the end of the list, using setNext
- At the start of the list, creating a new node with the old list as its next
- In the middle of the list, by rearranging links
- Bounded only by available memory

A basic operation with these lists is traversal

- Visiting each node in the list, one at a time
- Can be with a loop, checking for null as the continuation condition
- Can be with recursion, checking for null to signal the base case

Exercise 4.1.3. Add a static method printList to your NumberNode class of Exercise 4.1.1. It should take one NumberNode argument, and print a list of numbers, with nice punctuation before, after and between each number. So for example, the output for the list you built in Exercise 4.1.2 should be {1,2,3}
Exercise 4.1.4. Reimplement method `printList` as an object method using recursion. You will probably need to write a private recursive helper method, with `printList` making one call to the helper.

Exercise 4.1.5. Extend Exercise 4.1.4 with a method `insertAfter` which
- Takes a single int argument
- Returns no result
- Changes the list to insert the given number into the list after the current node

So for example, if `l` is a local reference to the first node of the list described in Exercise 4.1.2 then the result of running this code:

```java
l.getNext().insertAfter(4);
l.printList();
```

should be

```
[1,2,4,3]
```

Exercise 4.1.6. Extend Exercise 4.1.4 or 4.1.5 with a method `removeAfter` which
- Takes no argument and returns no result
- Changes the list to remove the one node following the current node from the list

So for example, if `l` is a local reference to the first node of the list described in Exercise 4.1.2 then the result of running this code:

```java
l.removeAfter();
l.printList();
```

should be

```
[1,3]
```

Exercise 4.1.7. Extend the `NumberNode` class with a method `get` which
- Takes a single int argument `n`, and
- Returns the `n`th element of the list, numbered from 0

Write both a recursive and a loop-based version of your method.

Exercise 4.1.8. Extend the `NumberNode` class with a method `length` which
- Takes no arguments, and
- Returns the number of nodes in the list, counting this node as 1.

Write both a recursive and a loop-based version of your method.

Exercise 4.1.9. Extend the `NumberNode` class with a method `newReversed`, which
- Takes no arguments, and
- Returns a list which contains the same elements as this, but in the reverse order. Your method must not alter this or any of the nodes to which it points, but should instead create new `NumberNode` instances.

Write both a recursive and a loop-based version of your method.
Exercise 4.1.10. Extend the `NumberNode` class with a method `reverse`, which

- Takes no arguments,
- Updates the next fields of `this` and the nodes to which it points so that the list is reversed, and
- Returns a reference to the first node of the new list.

Write both a recursive and a loop-based version of your method.

The following exercise uses the `IntegerPredicate` interface and `IntegerEvenPredicate` class of Exercise 2.2.1.

Exercise 4.1.11. Extend the `NumberNode` class with a method `filter`, which

- Takes one argument, an instance of `IntegerPredicate`, and
- Returns the first node of a linked list of new `NumberNode` instances which contain exactly the elements of `this` for which the predicate’s `check` method returns `true`.

Your `filter` method must not change either field of `this`, nor of any node to which `this` points. So for example, if `l` is a local reference to the first node of the list described in Exercise 4.1.2, then the result of running this code:

```java
l.filter(new IntegerEvenPredicate()).printList();
```

should be

[2]

The following two exercises use the `IntegerFunction` interface and `IntegerSquareFunction` class of Exercise 2.2.3.

Exercise 4.1.12. Extend the `NumberNode` class with a method `map`, which

- Takes one argument, an instance of `IntegerFunction`, and
- Returns the first node of a linked list of new `NumberNode` instances which contain exactly the result of applying the argument’s `apply` method to the respective element of `this`.

Your `map` method must not change either field of `this`, nor of any node to which `this` points. So for example, if `l` is a local reference to the first node of the list described in Exercise 4.1.2, then the result of running this code:

```java
l.map(new IntegerSquareFunction()).printList();
```

should be

[1, 4, 9]

The next exercise also uses the `IntegerBinaryFunction` interface and various classes of Exercise 2.2.5.
Exercise 4.1.13. Extend the `NumberNode` class with a method `reduce`, which

- Takes two arguments,
  1. An instance of `IntegerBinaryFunction` (which we will call `f`), and
  2. An integer (which we will call `z`).
- Returns the integer arising from using `f`'s `apply` method to combine the integers of `this` into a single integer, one element at a time, as follows:
  - `reduce` takes the numbers in order.
  - When `reduce` acts on an element of the list, it uses that number as the second argument to `apply`, and the result for the previous elements of the list for the first argument to `apply`.
  - To act on the first element of the list, it uses `z` as the first argument to `apply`.

Your `reduce` method must not change either field of `this`, nor of any node to which `this` points. So for example, if `l` is a local reference to the first node of the list described in Exercise 4.1.2, then

- The result of running
  ```java
  System.out.println(l.reduce(new IntegerAddition(), 0));
  ```
  should be 6.
- The result of running
  ```java
  System.out.println(l.reduce(new IntegerAddition(), 1000));
  ```
  should be 1006.
- The result of running
  ```java
  System.out.println(l.reduce(new IntegerMultiplication(), 1));
  ```
  should be 6.
- The result of running
  ```java
  System.out.println(l.reduce(new IntegerTenTwist()), 0);
  ```
  should be 123.

Hiding implementation details

The user of a linked list should not have to worry about how the list is implemented

- They should only need to deal with the actual data in the list
- Can we hide the `NumberNode` class entirely?
### Nested classes

A nested class is a class defined within another class

- There are two kinds: static nested classes and inner classes

```java
public class OuterClass {
    //...
    private static class StaticNestedClass {
        //...
    }
    private class InnerClass {
        //...
    }
}
```

- Useful for organizing code and keeping related classes together
- Hides unnecessary details from other classes
- Makes testing harder
  - Can no longer create `NumberNode` objects by themselves for test purposes

### Inner classes

- Associated with one instance of the outer class
- Can access all methods and data of its instance the outer class
  - Regardless of their accessibility (public or private)
- Can be marked private and completely hidden from other classes

```java
public class NumberList {
    private NumberNode head, tail;
    private int size;
    //...
    private class NumberNode {
        private int num;
        private NumberNode next, prev;
        //...
    }
}
```
4.1.1 TODO — Make into exercise(s) on sorted lists

Sorted lists

Once we can compare our objects, we can ensure that our own lists are *sorted* by changing how objects get added.

```
<table>
<thead>
<tr>
<th>Car to add</th>
</tr>
</thead>
<tbody>
<tr>
<td>make: Honda</td>
</tr>
<tr>
<td>vin : 555</td>
</tr>
</tbody>
</table>
```

Implementation of sorted insert

Sorted insertion is relatively simple, as long as the list is kept sorted

```
public void add(final Vehicle v) {
    VNode pos = head.next;
    while (pos != tail && v.compareTo(pos.data) >= 0) {
        pos = pos.next;
    }
    addBefore(pos, v);
}
```

Prevent users from messing up the ordering

```
public void add(final int idx, final Vehicle v) {
    // Ignore index specified by user
    add(v);
}
```

Maintaining unique items

Sorted lists can be used to maintain a unique set of items

- The list will hold no duplicates

```
public void add(Vehicle v) {
    VNode pos = head.next;
    while (pos != tail && v.compareTo(pos.data) > 0) {
        pos = pos.next;
    }
    // v.compareTo(pos.data) <= 0, so check equality
    if (pos != tail && v.equals(pos.data)) {
        return false; // Don't add if already stored
    }
    addBefore(pos, v);
}
```

Determining the size of a list

A list can be *traversed* by starting at the head node and following the next links to the end, accessing data along the way.
• TODO: Implement the size method

```java
public int size() {
    int numNodes = 0;
    NumberNode pos = head;
    while (pos != null) {
        ++numNodes;
        pos = pos.getNext();
    }
    return numNodes;
}
```

This requires \( O(n) \) time! Can we do better?

• All mutations to the chained nodes goes through NumberList
• So just add a size field to the NumberList class

The revised list class diagram

The list can keep track of its size internally.

```
NumberList
- int size - NumberNode head
  «constructor» + NumberList(int)[0.5em] + int size() +
  String toString()[0.5em] «update» + void set(int, int)
  «constructor» + NumberNode{int}[0.5em] + int getNum() +
  NumberNode getNext()[0.5em] «update» + void setNum(int) +
  void setNext(NumberNode)
```

Managing linked lists: sentinel nodes

There is an irritating inconsistency in the structure of these lists

• When we have an empty list with no NumberNode objects, the head field of NumberList is null
• Not a problem so far

• But when we insert elements into a list, we will sometimes need to detect a null head and update it to a NumberNode instance

• And when we delete elements into a list, we will sometimes need to detect a non null head and update it to null

The solution is to add an extra NumberNode at the beginning of the chain

• Called a sentinel node

• The sentinal holds no data, just points to the first actual node of the list
Managing the head node

The head node is the only node referenced directly by the list itself through a specific variable

• Every other node is referenced indirectly through a next pointer
• Losing the head node and/or "severing" its pointer can destroy all data in the list

```java
public class NumberList {
    private NumberNode head;
    //...
}
```

4.1.2 **TODO** Make these into (singly-linked) exercises

A for loop for list nodes

We can also use a for loop to iterate through a doubly-linked list

• Start at one sentinel node and proceed until we find the other

```java
public int indexOfAlt(int value) {
    int idx = 0;
    for(NumberNode pos=head.next; pos!=tail; pos=pos.next) {
        if (pos.data == value) {
            return idx;
        }
        ++idx;
    }
    return -1;
}
```

Can we do better?

• In terms of $O(n)$, no
• But we can reduce the number of actions for each node

A while loop for list nodes

Let’s go back to `indexOf`

• Could you think of a way to make the loop quicker?
• Hint: how can we avoid testing for the tail sentinel at every node?
• Remember that the sentinels are regular `NumberNode` instances
  – They have a data field
  – We just choose to disregard it
• So we can put whatever we want in the sentinel’s data field
  – For example, we could put `value` there
  – Now we know that we will find `value` in the `NumberNode` chain
We need to check just once that we are not at the tail sentinel

```java
public int indexOfAlt(int value) {
    int idx = 0;
    tail.data = value;
    NumberNode pos = head.next;
    while (pos.data != value) {
        pos = pos.next;
        ++idx;
    }
    if (pos == tail) {
        return -1;
    } else {
        return idx;
    }
}
```

Creating a list of vehicles
Let’s modify the NumberList so that it can store Vehicle objects instead.

- NumberNode becomes VNode
- Type for data: int becomes Vehicle
- == becomes .equals

All classes extend Object, and so they inherit an equals method:

```java
object1.equals(object2)
```
- By default, equals is defined as ==, which returns true if and only if the two objects occupy the same location in memory

4.2 Stacks and queues
Ordered data structures
Basic linked lists allow us to:

- Collect a number of objects
- Access them via get and iterators
- Add and remove items

For an unsorted list, there are no constraints on these operations

But sometimes, constraints on data insertion and access are desirable

- Netflix queue
- Amazon wishlist
- Email inbox
- Browser history
- Edit history in a word processor
Stacks
Recall:

• An abstract data type is a model that defines data types in terms of their behavior (what can be done with it)

A stack is an abstract data type that serves as a collection of elements with two principal operations:

• Push adds an item to the top of the stack

• Pop removes an item from the top of the stack

• Elements inserted into a stack come out in the reverse order

• A stack is a last-in-first-out (LIFO) structure

Stacks in the real world

• Pez dispensers

• A stack of textbooks

• A stack of dishes

Basic stack operations

The stack ADT provides methods to

• Check if stack is empty

• Access the top object

• Push onto the top of the stack

• Pop off the top of the stack

Consider a stack of strings

<table>
<thead>
<tr>
<th>StringStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-constructor-&gt; +</td>
</tr>
<tr>
<td>StringStack(){}</td>
</tr>
<tr>
<td>&lt;-query-&gt; + boolean isEmpty() +</td>
</tr>
<tr>
<td>String top(){}</td>
</tr>
<tr>
<td>&lt;-update-&gt; + void push(String) + String pop()</td>
</tr>
</tbody>
</table>
Implementing the stack

How should we actually implement the stack? A template as a starting point:

```java
public class StringStack {
    public StringStack() {
        // FILL IN
    }
    public boolean isEmpty() {
        return false; // FILL IN
    }
    public String top() {
        return null; // FILL IN
    }
    public void push(String s) {
        // FILL IN
    }
    public String pop() {
        return null; // FILL IN
    }
}
```

Implementing the stack via lists

Use a `GenericList<String>` behind the scenes

```java
public class StringStack {
    // Hidden from the stack user
    private final GenericList<String> list = new GenericList<String>();
    public StringStack() {
    }
    public boolean isEmpty() {
        return false; // FILL IN
    }
    public String top() {
        return null; // FILL IN
    }
    public void push(String s) {
        // FILL IN
    }
    public String pop() {
        return null; // FILL IN
    }
}
```

Implementing the stack

The `push` and `pop` methods will be the only way to modify the internal list

```java
public class StringStack {
    // Hidden from the stack user
    private final GenericList<String> list = new GenericList<String>();
    public StringStack() {
    }
    public boolean isEmpty() {
        return false; // FILL IN
    }
    public String top() {
        return null; // FILL IN
    }
    public void push(String s) {
        // FILL IN
    }
    public String pop() {
        return null; // FILL IN
    }
}
```
public String top() {
    return list.get(0); // Always constant time
}
public void push(String s) {
    list.add(0, s);
}
public String pop() {
    return list.remove(0);
}

New slide A list field, or a list superclass?

Implementing a generic stack

As with our generic list, to make a generic stack it suffices to just parameterize the content type

<table>
<thead>
<tr>
<th>StringStack</th>
<th>GenericStack&lt;SomeType&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>«constructor» + StringStack() [0.5em]</td>
<td>«constructor» + GenericStack&lt;SomeType&gt;()</td>
</tr>
<tr>
<td>«query» + boolean isEmpty() + String top() [0.5em]</td>
<td>«query» + boolean isEmpty() + SomeType top()</td>
</tr>
<tr>
<td>«update» + void push(String) + String pop()</td>
<td>«update» + void push(SomeType) + SomeType pop()</td>
</tr>
</tbody>
</table>

Using the generic stack

Executing code:

GenericStack<String> stack = new GenericStack<String>();
for (int i = 0; i < 5; ++i) {
    stack.push("Word " + i);
    System.out.println("Top: " + stack.top());
} while (!stack.isEmpty()) {
    System.out.println("Pop: " + stack.pop());
}

we get output:

Top: Word 0
Top: Word 1
Top: Word 2
Top: Word 3
Top: Word 4
Pop: Word 4
Pop: Word 3
Pop: Word 2
Pop: Word 1
Pop: Word 0
Queues

A queue is an abstract data type that serves as a collection of elements with two principal operations:

- **Enqueue** adds an item to the back of the queue
- **Dequeue** removes an item from front of the queue
- Elements inserted into a queue come out in the *same order*
- A queue is a *first-in-first-out* (FIFO) structure

**Basic queue operations**

The *Queue* ADT provides methods to:

- Check if queue is *empty*
- Access the *front* object
- *Enqueue* to add to the back
- *Dequeue* to remove from the front

Consider a queue of strings

- Just as for the *StringStack*, we can implement using a private *GenericList*

<table>
<thead>
<tr>
<th>StringQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public final StringQueue() { }</td>
</tr>
<tr>
<td>Public boolean isEmpty()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GenericQueue&lt;SomeType&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private final GenericList&lt;SomeType&gt; list = new GenericList&lt;SomeType&gt;();</td>
</tr>
<tr>
<td>Public GenericQueue&lt;SomeType&gt;()</td>
</tr>
</tbody>
</table>

**Towards a generic queue**

To make a generic queue, we just need to *parameterize* the type

```java
public class GenericQueue<SomeType> {
    private final GenericList<SomeType> list = new GenericList<SomeType>();
    public GenericQueue<SomeType>() { }
    public boolean isEmpty() { }
    public SomeType dequeue() { }
    public void enqueue(SomeType item) { }
}
```

**Implementing a generic queue**

We hide the internal list from the user again:
return list.isEmpty();
}
public SomeType front() {
    // Index 0 is always constant time
    return list.get(0);
}
public void push(SomeType s) {
    // Append to end
    list.add(s);
}
public SomeType pop() {
    // Remove from front
    return list.remove(0);
}

Using the generic queue

```
final GenericQueue<String> queue = new GenericQueue<String>();
for(int i=0; i<5; ++i) {
    queue.enqueue("Word " + i);
    System.out.println("Front: " + queue.front());
}
while (!queue.isEmpty()) {
    System.out.println("Dequeue: " + queue.dequeue());
}
```

Front: Word 0
Front: Word 1
Front: Word 2
Front: Word 3
Front: Word 4
Dequeue: Word 0
Dequeue: Word 1
Dequeue: Word 2
Dequeue: Word 3
Dequeue: Word 4

Bounded stacks and queues

We can limit the number of entries in a stack or queue by just checking if the container is full before adding any items; if it is, throw an exception!

```
4.3 Doubly-linked lists

4.3.1 Double-linked lists

Types of linked list

- Linked lists where each node contains a single link to another node in the list

There are also *doubly linked lists*

- Linked lists where each node contains two links, one to the subsequent node and one to the preceding node

**Sentinel nodes in doubly linked lists**

Many implementations of doubly linked lists will use two sentinel nodes: one at the head, and one at the *tail*

Creating a doubly linked list
Initializing a doubly linked list

public class NumberList {
    private int size;
    private NumberNode head;
    private NumberNode tail;

    public NumberList() {
        clear();
    }

    public void clear() {
        // TODO: Write me
    }

    //...
}

NumberNode
- int num - NumberNode next -
  NumberNode prev
+ NumberNode(int, NumberNode, NumberNode)

Initializing a doubly linked list

public class NumberList {
    private int size;
    private NumberNode head;
    private NumberNode tail;

    public NumberList() {
        clear();
    }

    public void clear() {
        head = new NumberNode(0, null, null);
        tail = new NumberNode(0, head, null);
        head.next = tail;
        tail.prev = prev;
        size = 0;
    }

    //...
}
4.3.2 TODO  Make these into exercises

Accessing linked list elements
How would we implement the get method?

```java
public int get(int idx) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    return getNode(idx).data;
}
```

- User-accessible method that allows access to data stored in the list
- Internally, it finds the node at the specified position via the getNode helper method and then returns its data
- The getNode method is completely hidden from the user

Accessing linked list nodes
What’s the most efficient way to implement getNode?

```java
private NumberNode getNode(int idx) {
    // Allow retrieving the tail sentinel
    if (idx < 0 || idx > size) {
        throw new IndexOutOfBoundsException("Error!");
    }
    NumberNode node;
    if (idx < size / 2) { // Scan from front
        node = head.next;
        for(int i=0; i<idx; ++i) {
            node = node.next;
        }
    } else { // Scan from back
        node = tail;
        for(int i=size; i>idx; --i) {
            node = node.prev;
        }
    }
    return node;
}
```
Adding elements to a linked list

The user has two methods for adding elements:

• **add(int)** takes a value to be added at the end of the list

  ```java
  NumberList list = new NumberList(); // ()
  list.add(10); // (10)
  list.add(20); // (10, 20)
  list.add(30); // (10, 20, 30)
  ```

• **add(int, int)** takes an index and a value, and should insert the value into the list at the given index

  ```java
  NumberList list = new NumberList(); // ()
  list.add(0, 10); // (10)
  list.add(0, 20); // (20)
  list.add(1, 30); // (20, 30)
  list.add(1, 60); // (20, 60)
  list.add(3, 40); // (20, 30, 10, 40)
  list.add(3, 50); // (20, 30, 10, 40, 50)
  list.add(1, 60); // (20, 60, 30, 10, 50, 40)
  ```

Implementing element addition

Next: the two `add` methods

```java
public void add(int value) {
    add(size, value); // Insert before node at index ‘size’
}

public void add(int idx, int value) {
    // Find the node itself, and insert a new node before it!
    addBefore(getNode(idx), value);
}
```

• We need to create a new `NumberNode` which contains the given value, and then position it correctly within the list by updating links.

Implementing node addition

How do we implement the `addBefore` method?

```java
private void addBefore(NumberNode pos, int value) {
    NumberNode newNode = new NumberNode(value, pos.prev, pos);
    pos.prev = newNode;
    newNode.prev.next = newNode;
    ++size;
}
```

Implementing node removal

How do we implement the `remove` method for `NumberNode`s?

```java
private int remove(NumberNode pos) {
    pos.next.prev = pos.prev;
    pos.prev.next = pos.next;
    pos.next = null;
    pos.prev = null;
    --size;
    return pos.data;
}
```
Implementing element removal
With the previous methods as helpers, the remove method for elements is straightforward

```java
public int remove(int idx) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!"鞥n);
    }
    return remove(getNode(idx));
}
```

- These methods are an exception to the rule that we should separate query methods and mutator methods
  - Returning what we remove is a common pattern for deletions from data structures
  - But the exceptions prove the rule!

Implementing update methods
We can again use getNode for the set method

```java
public int set(final int idx, final int value) {
    if (idx < 0 || idx >= size) {
        throw new IndexOutOfBoundsException("Error!"鞥n);
    }
    NumberNode pos = getNode(idx);
    int oldVal = pos.data;
    pos.data = value;
    return oldVal;
}
```

Implementing the query methods
How do we implement the indexOf method?

```java
public int indexOf(final int value) {
    int idx = 0;
    NumberNode pos = head.next;
    while (pos != tail) {
        if (pos.data == value) {
            return idx;
        }
        ++idx;
        pos = pos.next;
    }
    return -1;
}
```
5 Generics

5.1 Generic classes

Generic data structures
It would be inconvenient to rewrite MyList every time we want to use a new type of data

- Goal: Write our list code in such a way so that it works with any complex data type

One possible solution: use Object for the list element type

- Simple and it works for any complex type
- Used in early versions of Java (1.0-1.4)
- Requires casting

```java
public class MyList {
    private Object[] items;
    public void add(final String item) {
        // ...
    }
}
```

// Use like this
final String fifth = (String)(lst.get(4));

   - Tedious
   - Can introduce errors in code — which we do not discover until runtime

Searching for a better solution
Consider how we write and use a method:

- We specify formal parameters that represent inputs to the method
- We write the method so that it works regardless of what those inputs actually are
- When we call (invoke, use) the method, we pass in actual values (arguments) for it to process.

We would like to be able to do something similar for a class:

- Specify parameters that represent type "inputs" to the class
- Write the class so that it works regardless of what those type inputs actually are
- Provide actual types (arguments) for the type "inputs" when we use the class (at variable declarations and object instantiations)

Java generics
Generic types provide a way to do this by using the concept of a type variable

- Added to Java 5 in 2004
- Allows for classes and methods to be written for any complex type

Specifically, generics allow non-primitive types to be type parameters when defining classes, interfaces, and methods

A generic class is a class that is defined with one or more type parameters (type variables). (A class that takes “inputs”.)
public class MyGenericClass<T1, T2, ..., Tn> { /* .. */ }

More info at [https://docs.oracle.com/javase/tutorial/java/generics/types.html](https://docs.oracle.com/javase/tutorial/java/generics/types.html)

An analogy with methods

- A method specifies input values via formal parameters
  
  ```java
  public static void someMethod(int var1, String var2) {
    /* .. */
  }
  ```
  
  - `var1` and `var2` are formal parameters for `someMethod`
  - Can be used anywhere in the method itself where an `int` or a `String` would be used

- A generic class specifies type values via type parameters
  
  ```java
  public class MyGenericClass<TypeVar1, TypeVar2> { /* .. */ }
  ```
  
  - `TypeVar1` and `TypeVar2` are type parameters for the class
  - Can be used anywhere in the class itself where a `type` would be used

A simple example

A regular class

```java
public class Box {
  private Object data;
  public Box(Object d) {
    data = d;
  }
  public void set(Object d) {
    data = d;
  }
  public Object get() {
    return data;
  }
}
```

- Defines a type called `Box`

A generic class

```java
public class Box<ContentType> {
  private ContentType data;
  public Box(ContentType d) {
    data = d;
  }
  public void set(ContentType d) {
    data = d;
  }
  public ContentType get() {
    return data;
  }
}
```

- Defines a generic type which requires an input in order to be used
Generic type invocation

• Method invocation passes arguments (values) to a method

```java
generic type invocation
public static void someMethod(int var1, String var2) { /* ... */ }
generic type invocation
public static void main(String[] args) {
    someMethod(42, "Hello, World!");
}

– Method is run using 42 and "Hello, World!" for var1 and var2

• Generic type invocation passes type arguments to a generic class

```java
generic type invocation
public static void main(String[] args) {
    // Create parameterized type
    final Box<String> wordContainer;
    // Instantiate
    wordContainer = new Box<String>("Hello, World!");
    // Another type and instance
    final Box<Vehicle> carBox = new Box<Vehicle>(/* ... */);
}

– Creates the parameterized types Box<String> and Box<Vehicle>
– Instantiates objects of these parameterized types

Empty angle brackets

• In Java 5 and 6, instantiating a parameterized class requires that the type argument appear twice

```java

• From Java 7 this was simplified

```java

– Only the first use of the type argument is necessary

```java

– The empty angle brackets <> are sometimes referred to as the diamond operator
– Both styles work in recent versions of Java
– The second style is preferred
– Must use the first if dealing with legacy code

Creating a generic list class

We can make IntegerList or VehicleList generic by adding a type parameter and revising the code (and tests) accordingly

• General class GenericList with type variable ContentType

• Any method dealing with the data replaces Vehicle with the GenericList class’s type parameter ContentType

• Any method that doesn’t use the data stays exactly the same

From VehicleList...
public class VehicleList {
    public void add(int idx, Vehicle v) {
        addBefore(getNode(idx), v);
    }
    public void addBefore(Node p, Vehicle v) {
        final Node newNode = new Node(v, p.prev, p);
        newNode.prev.next = newNode;
        p.prev = newNode;
        ++size;
    }
    // ...
}

...to GenericList

public class GenericList<ContentType> {
    public void add(int idx, ContentType e) {
        addBefore(getNode(idx), e);
    }
    public void addBefore(Node p, ContentType e) {
        final Node newNode = new Node(e, p.prev, p);
        newNode.prev.next = newNode;
        p.prev = newNode;
        ++size;
    }
    // ...
}

Using a generic list
When we instantiate a generic list, we must provide a type argument

GenericList<Vehicle> lot = new GenericList<Vehicle>();
GenericList<Integer> numbers = new GenericList<>();

- In the second line Java can work out the argument type from other information — but that information must be present
- Adding an object to a generic list using add or set is checked by the compiler to ensure type compatibility
- Accessing an object from a generic list using get or remove carries the declared type of the list

Convert the next two from ArrayList to our list

GenericList<Integer> numbers = new GenericList<>();

Primitive and reference types
The ArrayList class only supports reference types.

- This does not work:

        ArrayList<int> list = new ArrayList<int>(); // INCORRECT!
Solution: another layer of abstraction!

- **Integer** class provides a wrapper for `int`
- **Double** class provides a wrapper for `double`
- And so on
- **All standard in java.lang**

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(42));
Integer first = list.get(0);
```

**Autoboxing and unboxing**

Having to create objects for each `int` we add to the list is cumbersome. We’d like to be able to do the following:

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(42);
int first = list.get(0);
```

In fact, we can do just that!

- **Autoboxing** is the process by which a primitive type is automatically converted to its corresponding wrapper object.
- **Unboxing** is the process by which a wrapper object is automatically converted back to its primitive type.

See also [docs.oracle.com/javase/tutorial/java/data/autoboxing.html](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html)

5.1.1 TODO Work in this set

Creating a generic list class

We can make `VehicleList` generic by adding a type parameter and revising the code (and tests) accordingly

- General class `GenericList` with type variable `ContentType`
- Any method dealing with the data replaces `Vehicle` with the `GenericList` class’s type parameter `ContentType`
- Any method that doesn’t use the data stays exactly the same

**From VehicleList...**

```java
public class VehicleList {
    public void add(int idx, Vehicle v) {
        addBefore(getNode(idx), v);
    }
    public void addBefore(Node p, Vehicle v) {
        final Node newNode = new Node(v, p.prev, p);
        newNode.prev.next = newNode;
        p.prev = newNode;
        ++size;
    }
    // ...
```
...to `GenericList`

```java
public class GenericList<ContentType> {
    public void
        add(int idx, ContentType e) {
            addBefore(getNode(idx), e);
        }
    
    public void
        addBefore(Node p, ContentType e) {
            final Node newNode
                = new Node(e, p.prev, p);
            newNode.prev.next = newNode;
            p.prev = newNode;
            ++size;
        }

    // ...
}
```

**Using a generic list**

When we instantiate a generic list, we must provide a type argument

```java
GenericList<Vehicle> lot = new GenericList<Vehicle>();
GenericList<Integer> numbers = new GenericList<>();
```

- In the second line Java can work out the argument type from other information — but that information must be present
- **Adding an object to a generic list using** `add` **or** `set` **is checked by the compiler to ensure type compatibility**
- **Accessing an object from a generic list using** `get` **or** `remove` **carries the declared type of the list**

**Iterating for a generic list**

The `Iterable` and `Iterator` interfaces share the type parameter of the list itself

```java
public interface Iterable<T> {
    public Iterator<T> iterator();
}

public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}
```

```java
public class GenericList<ContentType>
    implements Iterable<ContentType> {
    //...

    public Iterator<ContentType> iterator() {
        return new GenericListIterator();
    }

    private class GenericListIterator
        implements Iterator<ContentType> {
        //...
        public ContentType next() {
            pos = pos.next;
        }
    }
```
5.2 Generic methods

We've seen classes parametrized over types

```java
public class Box<T> {
    private T data;
    public Box(T d) {
        data = d;
    }
    public void set(T d) {
        data = d;
    }
    public T get() {
        return data;
    }
}
```

- Our analogy was to compare methods taking value arguments with classes taking type arguments
- But if a method can take a value argument, why could it not take a type argument as well?

**Type parameters**

In fact, methods can take both type and value arguments

```java
public <T1, ..., Tn> ResultType method(ArgType1 arg1, ... ArgTypeN argN) {
    ...
}
```

- The method must work when substituting any types for the T1, ..., Tn
- The type variables T1, ..., Tn can appear in the ResultType as well as the ArgType's
- The T1, ..., Tn are separate from any type variables associated with the class
  - In fact the class may be monomorphic

**Removing one list element**

```java
public <A> void shorten(List<A> orig) {
    if (orig.size()>0) {
        orig.remove(0);
    }
}
```

- Works for any type A
- Checking the length and removing the first element apply to any element type
Type parameter in the result type

```java
public <A> List<A> pull(List<A> orig) {
    final List<A> result = new ArrayList<>();
    if (orig.size()>0) {
        result.add(orig.get(0));
        orig.remove(0);
    }
    return result;
}
```

- We do not need to know the actual value of A to create a list to hold A values

Two type variables

```java
public abstract class PolyClass<P> {
    public abstract boolean tester(P p);
    public <Q> boolean check(Q item1, P item2) {
        if (tester(item2)) {
            return false;
        } else {
            return item1.toString().length() % 2 == 1;
        }
    }
}
```

- P and Q are not necessarily the same type
  - But they are also not necessarily different
  - Since we can make no assumptions about the type of Q, only methods defined for all objects are valid for arguments of declared type Q

Exercise 5.2.1.  [TODO]

Exercise 5.2.2. In Exercise 2.2.12 we used abstract methods to deploy the Template Method pattern. We can achieve the same structure by passing instances of java.util.function types to the constructor of a Template Method base class.

Write a concrete class IntervalConsumingCalculation with

- A single constructor which takes four arguments:
  1. An instance of Function<Integer, Integer> named intervalStartMaker
  2. An instance of Function<Integer, Integer> named intervalEndMaker
  3. An instance of Supplier<Long> named initialResultSupplier
  4. An instance of BiFunction<Long, Int, Long> named nextResultCalculator

- A single method performCalculation, which makes the same steps as in Exercise 2.2.12 but using calls on the objects passed to the constructor instead of calls to the abstract methods.

Text for java.util.function interface examples

The standard Java library contains a number of interesting interfaces; the next few interfaces introduce you to several of them. Remember that when your program uses classes which are not in package java.lang, you will need to explicitly import them.
Then write new versions of the two classes DerivedFactorial and DerivedSumThrough which extend IntervalConsumingCalculation instead of AbstractIntervalConsumingCalculation. Just as before, your revised DerivedFactorial and DerivedSumThrough classes must not override the performCalculation method! Moreover, they should not introduce new methods. Their constructors should take no arguments, and they should simply pass appropriate java.util.function instances in their constructors’ calls to super.

5.3 The Iterable interface

Interfaces with type parameters

Interfaces can also take type parameters

• You may have encountered the method iterator

• Gives an object which lets us see the elements of an array or list one at a time

```java
final String[] myStrings;
// Setup for myStrings omitted

final Iterator<String> iter = myStrings.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

5.3.1 TODO Start of one iteration "introduction"

The VehicleList class

Let’s turn back to the unsorted list

We can:

• Create a list

• Add to list

• Remove from list

• Modify an element

• Find an element

• Get length

What else might we want to do?

• Loop over it!
Looping through a list
If we use the same loop as for array-based lists, we lose

```java
for (int i=0; i<list.size(); ++i) {
    System.out.println(list.get(i));
}
```
Each call to `get` is $O(n)$, so the whole traversal would be $O(n^2)$ — not acceptable

Internally, a list can loop through itself because it has direct access to head, tail, and the data nodes

- These details are private, so other classes cannot access them

**Solution:** The list can provide other classes with an iterator

- An iterator is an object that enables a programmer to traverse a container, without needing to know the underlying structure of the container itself

```java
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```
```java
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}
```

Using a list iterator

How does list iteration work?

- **Internal iteration** in the `VehicleList` class

```java
public void display() {
    VNode pos = head.next;
    while (pos != tail) {
        System.out.println(pos.data);
        pos = pos.next;
    }
}
```

- **External iteration** by an outside class

```java
VehicleList list = new VehicleList();
// ... add items to list ...

final Iterator iter = list.iterator();
// Just like Scanner: get next and advance
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

**Step 1: Make the list Iterable**
A container must indicate that it can be iterated over by implementing the `Iterable` interface

---

125
public class VehicleList implements Iterable<Vehicle> {
    // ...
    public Iterator<Vehicle> iterator() {
        // TODO: We’ll fill this in later
        throw new UnsupportedOperationException();
    }
}

• The iterator method must return an object that can do the things an iterator should be able to do
  – Namely, an object that implements the Iterator interface
• We can specify a template for such an object as an inner class

Step 2: Make an Iterator
The class of iterator objects is often defined as a private inner class of the container itself

public class VehicleList
    implements Iterable<Vehicle> {
    // ...
    public Iterator<Vehicle> iterator() {
        return new VIterator();
    }
    private class VIterator
        implements Iterator<Vehicle> {
        public boolean hasNext() {
            // TODO: We’ll fill this in later
            throw new UnsupportedOperationException();
        }
        public Vehicle next() {
            // TODO: We’ll fill this in later
            throw new UnsupportedOperationException();
        }
        public void remove() {
            // TODO: We’ll fill this in later
            throw new UnsupportedOperationException();
        }
    }
}

• The VIterator class defines a template for iterators over a VehicleList
• The Iterable container will provide an Iterator object when asked for one

Building the VIterator class
In order to iterate through a VehicleList, a VIterator object needs to know where it is in the list at any point in time

public class VehicleList implements Iterable<Vehicle> {
    // ...
    private class VIterator implements Iterator<Vehicle> {
        private VNode pos;
        public VIterator() {
            pos = head;
        }
    }
}
public boolean hasNext() {
    return (pos.next != tail);
}

public Vehicle next() {
    if (!hasNext()) {
        throw new NoSuchElementException("...");
    }
    pos = pos.next;
    return pos.data;
}

public void remove() {
    throw new UnsupportedOperationException();
}

• Track the current position as pos
• Starts at head sentinel
  – Keep pos before the node whose content would be returned for a call to next()
• We can tell if there is another element by whether pos.next is the tail sentinel
• At each call to next(), advance pos and return data
• We’ll opt not to support remove

Using the iterator
The iterator allows external classes to loop over a list’s contents

final VehicleList list = new VehicleList();
/* add contents to list here */

int hondaCount = 0;

final Iterator<Vehicle> iter = list.iterator();
while (iter.hasNext()) {
    final Vehicle nextVehicle = iter.next();
    if (nextVehicle.getMakeModel().equals("Honda Civic")) {
        ++hondaCount;
    }
}

System.out.println("There are " + hondaCount + " Honda Civics!");

The for-each loop
The for-each loop can be used with any Iterable container
• With our VehicleList

for (final Vehicle v : list) {
    System.out.println(v.getMakeModel());
}

127
Read this as "for each \( v \) in list"

- With arrays

```java
int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (final int x : arr) {
    System.out.println("Value: "+ x);
}
```

### 5.3.2 TODO Another iteration "introduction"

**Two generic interfaces**

Under the hood, there are two generic interfaces behind this mechanism

#### The `Iterable` interface

```java
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

#### The `Iterator` interface

```java
public interface Iterator<T> {
    public T next();
    public boolean hasNext();
}
```

### Making `MyList` iterable

First, of course, a test

```java
@Test public void iterator() {
    final MyList lst = new MyList();
    lst.add("thing1");
    lst.add("thing2");

    final Iterator<String> iter = lst.iterator();
    assertEquals("First element iterated first",
                "thing1", iter.next());
    assertEquals("Second element iterated next",
                "thing2", iter.next());
    assertFalse("Nothing else was in list",
                iter.hasNext());
}
```

#### A class for the iterator

```java
class MyListIterator implements Iterator<String> {
    private int index=0;
    private final int limit;
    private final String[] strings;
    public MyListIterator(int limit, String[] strings) {
        this.limit = limit;
        this.strings = strings;
    }
```
public String next() {
    return strings[index++];
}
public boolean hasNext() {
    return index<limit;
}

Putting it all together

public class MyList
    implements Iterable<String> {
    // Rest unchanged
    public Iterator<String> iterator() {
        return new MyListIterator(used, strings);
    }
}

• And the test passes

Short-form for loop

• The Iterable class is behind the short-form for loop

• Any object of a class implementing Iterable can be used in these loops:

    for (final String s : myList) {
        System.out.println(s);
    }

From later — work in

Iterating for a generic list

The Iterable and Iterator interfaces share the type parameter of the list itself

public interface Iterable<T> {
    public Iterator<T> iterator();
}

public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}

public class GenericList<SomeType>
    implements Iterable<SomeType> {
    //...
    public Iterator<SomeType> iterator() {
        return new GenericListIterator();
    }
    private class GenericListIterator
        implements Iterator<SomeType> {

public SomeType next() {
    pos = pos.next;
    return pos.data;
}

5.4 The Comparable interface

Another look at Comparable

Comparable is actually a generic interface

- Specify the type to which it is valid to compare

public interface Comparable<T> {
    public int compareTo(T o);
}

- So we can compare an Integer to an Integer, but not to a String

Sorted lists

Suppose we want to keep our list of Vehicle instances sorted in some way.
- Need a way to compare two Vehicle instances and identify which comes first

We can implement the Comparable interface

public interface Comparable<T> {
    public int compareTo(T obj);
}

The compareTo() method should:
- Return 0 if the invoking object and obj are "equal"
- Return a negative number if invoking object is "less than" obj
- Return a positive number if invoking object is "greater than" obj

Implementing the Comparable interface

As with equals, the programmer defines what equal-to, greater-than, and less-than mean

For example, we will:
- Sort Car instances first by make/model, then by mileage
- Sort Truck instances by make and model, then capacity, then mileage
- Sort Van instances by make and model, then number of passengers, then mileage

We can ensure that all subclasses provide an implementation of compareTo by having the Vehicle class implement the interface

public abstract class Vehicle implements Comparable<Vehicle> {
    • So Vehicle agrees to provide a method int compareTo(Vehicle v)
    • But we do not define any such method in Vehicle!
    • It’s OK because Vehicle is abstract
      - We are placing an obligation on the concrete children of Vehicle
A first try at a `compareTo` method for Car

```java
public class Car extends Vehicle {
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            // Not a car, so unclear how to order
            return 0;
        }
        final Car c = (Car)other;

        // Compare makeModel first
        if (!makeModel.equals(c.makeModel)) {
            return makeModel.compareTo(c.makeModel);
        }

        // If same makeModel, compare by mileage
        return getMileage() - c.getMileage();
    }
}
```

Requirements for the `compareTo` method

The `compareTo` method should provide a *total ordering* on the objects of each class that implements it:

- **Sign should flip when reversing caller and argument**
  - If `(x.compareTo(y) < 0)` then `(y.compareTo(x) > 0)`
  - If `(x.compareTo(y) > 0)` then `(y.compareTo(x) < 0)`

- **It should be transitive**
  - If `(x.compareTo(y) < 0 && y.compareTo(z) < 0)` then `(x.compareTo(z) < 0)`

- **Sign should be consistent for equal objects**
  
  If `x.compareTo(y)` returns 0 then either:
  - `(x.compareTo(z) > 0 && y.compareTo(z) > 0)`, or
  - `(x.compareTo(z) < 0 && y.compareTo(z) < 0)`

See also the [Comparable Javadoc page](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)

**Consistency between `compareTo` and `equals`**

It is also *strongly recommended* that that `compareTo` and `equals` be consistent

`(x.compareTo(y) == 0) == (x.equals(y))`

That is: `compareTo` should say two objects are equal if and only if `equals` says that they are equal

So we should always have one of the scenarios below

- But it is *not* enforced by the compiler
- Up to the programmer to stay consistent

**Both equal**
obj1.equals(obj2) == true
obj1.compareTo(obj2) == 0

Both not equal

obj1.equals(obj2) == false
obj1.compareTo(obj2) != 0

An inconsistent compareTo and equals

public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        Car c = (Car) other;
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            return 0;
        }
        Car c = (Car) other;
        if (!this.makeModel.equals(c.makeModel)) {
            return this.makeModel.compareTo(c.makeModel);
        }
        return getMileage() - c.getMileage();
    }
}

• So is a Car instance equal to instances of other classes, or not?

Are these compareTo and equals consistent?

public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        final Car c = (Car) other;
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            return -1;
        }
        final Car c = (Car) other;
        if (!this.makeModel.equals(c.makeModel)) {
            return this.makeModel.compareTo(c.makeModel);
        }
        return getMileage() - c.getMileage();
    }
}
• We have ordered Car instances before non-cars

• Is this viable?

**More trouble with related classes**

The fix on the previous slide

```java
if (!(other instanceof Car)) {
    return -1;
}
```

works to ensure consistency when comparing Car instances, but also introduces some odd behavior if adopted as is for Truck

```java
public class Truck extends Vehicle {
    // ...
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            return -1;
        }
        // ...
    }
}
```

// And executing statements
Car c = new Car("Honda Civic", 214118);
Truck t = new Truck("Ford F-150", 0, 5);
if (c.compareTo(t) < 0) { System.out.println("c less than t"); }
if (t.compareTo(c) < 0) { System.out.println("t less than c"); }

This will print:
c less than t
t less than c

**Comparison across related classes**

We need to also enforce an ordering across classes

```java
public class Car extends Vehicle {
    // ...
    public int compareTo(Object other) {
        if (!(other instanceof Car)) {
            return -1; // Cars always come first
        }
        // ...
    }
}
```

```java
public class Van extends Vehicle {
    // ...
    public int compareTo(Object other) {
        if (!(other instanceof Van)) {
            return 1; // Vans always come last
        }
        // ...
    }
}
```

133
Comparison across related classes

We need to also enforce an ordering across classes.

public class Truck extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            if (other instanceof Car) {
                return 1; // Trucks come after Cars
            } else if (other instanceof Van) {
                return -1; // Trucks come before Vans
            } else {
                return -1; // Trucks before anything else
            }
        }
        // ...
    }
}

• Requires subclasses to know about each other
• Not particularly sustainable for a larger number of classes
• Burdensome when adding new subclasses of Vehicle

5.5 Type parameter bounds

Generic sorted lists

Recall our sorted list class:

public class SortedVehicleList {
    // ...
    public void add(final Vehicle v) {
        VNode pos = head.next;
        while (pos != tail
                && v.compareTo(pos.data) >= 0) {
            pos = pos.next;
        }
        addBefore(pos, v);
    }

Can we make it generic in the same way?

public class SortedList<T> {
    // ...
    public void add(final T item) {
        Node pos = head.next;
        while (pos != tail
                && item.compareTo(pos.data) >= 0) {
            pos = pos.next;
        }
        addBefore(pos, item);
    }

• No! We cannot just assume that any class T implements Comparable
Type variable bounds
We can impose bounds on type variables

• A simple example first:

```java
public interface Plastic {
    public String color();
}
```

• Then we can make a version of Box which holds only objects which conform to Plastic

```java
public class PlasticBox<T extends Plastic> {
    // ...just like Box...
}
```

• Java will enforce this bound when we describe a particular PlasticBox

```java
public class FoodWrap implements Plastic {
    // ...
}
public class AluminumFoil {
    // *Not* Plastic
    // ...
}
```

– This declaration compiles:

```java
PlasticBox<FoodWrap> p;
```

– This declaration is erroneous:

```java
PlasticBox<AluminumFoil> p;
```

Bounds and type arguments
Comparable is more complicated than Plastic

• Comparable requires a type argument

• What argument should it take?

  – In other words: what type of values do we want to compare to?

• When we think of comparison and total orders, we think about comparing similar things

  – We’d compare an Integer to an Integer
  – We’d compare a String to a String
  – We would not compare a String to an Integer

  – Remember the laws for these things!

• So the elements of a SortedList must be comparable to themselves

  – We could use the type variable again in the bound

```java
public class SortedList<T extends Comparable<T>> {
    // ...
```

  – Is this good enough?
With Vehicle
We could now define a sorted list of Vehicle using the generic type

• We’ve already made Vehicle implement Comparable<Vehicle>
• So Vehicle satisfies the bound
• Progress!

With Car
What if we wanted a SortedList<Car>?

• We can certainly use compareTo on two Car instances
• So we should be able to make a sorted list of them
• But Car not satisfy the bound!
  – Doesn’t Car implement Comparable<Car>?
  – No — Car implements Comparable<Vehicle>!

• So Car does not fit the pattern T implements Comparable<T>
  – And a declaration
    SortedList<Car> cars;
  will raise an error

Why would Car be reasonable?
Car's compareTo method can be applied to more than just Car instances

• SortedList would be fine with Car implementing Comparable<Car>

• But when Car implements Comparable<Vehicle>, we want SortedList to understand that the method is still OK for any Car
  – This is true because Car is a subclass of Vehicle
  – This would be true for any class for which Car is a subclass
  – Or in other words, for any superclass of Car

• We want to declare SortedList so that its type argument T
  – Can be compared to any superclass of T
  – We don’t care exactly what it’s comparable to, so long as it is a superclass of T

• Java lets us make such a declaration:

  public class SortedList<T extends Comparable<? super T>> {
    – So now, when we declare
      SortedList<Car> cars;
    Java will notice
      1. That the type argument Car implements Comparable<Vehicle>, and
      2. That Vehicle is in fact some superclass of the argument Car
    So we’ve satisfied the constraint!
From the Java standard library
At first these type bounds seem like a strange and exotic idea
But we find them in a number of places in the Java libraries

• In enumerated types

• In classes for event handling

• In methods for collections classes like SortedSet

(Screenshots from SDK Javadoc pages)

5.5.1 Comparables

Sorted lists
Suppose we want to keep our list of Vehicle instances sorted in some way.

• Need a way to compare two Vehicle instances and identify which comes first

We can implement the Comparable interface

```java
public interface Comparable<T> {
    public int compareTo(T obj);
}
```

The compareTo() method should:

• Return 0 if the invoking object and obj are “equal”
• Return a negative number if invoking object is "less than" obj
• Return a positive number if invoking object is "greater than" obj

Implementing the Comparable interface
As with equals, the programmer defines what equal-to, greater-than, and less-than mean

For example, we will:

• Sort Car instances first by make/model, then by mileage
• Sort Truck instances by make and model, then capacity, then mileage
• Sort Van instances by make and model, then number of passengers, then mileage

We can ensure that all subclasses provide an implementation of compareTo by having the Vehicle class implement the interface
public abstract class Vehicle implements Comparable<Vehicle> {

- So Vehicle agrees to provide a method \texttt{int compareTo(Vehicle v)}
- But we do not define any such method in Vehicle!
- It's OK because Vehicle is \textit{abstract}
  - We are placing an obligation on the concrete children of Vehicle

A first try at a \texttt{compareTo} method for Car

public class Car extends Vehicle {
    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            // Not a car, so unclear how to order
            return 0;
        }

        final Car c = (Car)other;
        // Compare makeModel first
        if (!makeModel.equals(c.makeModel)) {
            return makeModel.compareTo(c.makeModel);
        }

        // If same makeModel, compare by mileage
        return getMileage() - c.getMileage();
    }
}

Requirements for the \texttt{compareTo} method

The \texttt{compareTo} method should provide a total ordering on the objects of each class that implements it

- Sign should flip when reversing caller and argument
  - If \texttt{(x.compareTo(y) < 0)} then \texttt{(y.compareTo(x) > 0)}
  - If \texttt{(x.compareTo(y) > 0)} then \texttt{(y.compareTo(x) < 0)}

- It should be transitive
  - If \texttt{(x.compareTo(y) < 0 \&\& y.compareTo(z) < 0)} then \texttt{(x.compareTo(z) < 0)}

- Sign should be consistent for equal objects
  
  If \texttt{x.compareTo(y)} returns \texttt{0} then either:
  - \texttt{(x.compareTo(z) > 0 \&\& y.compareTo(z) > 0)}, or
  - \texttt{(x.compareTo(z) < 0 \&\& y.compareTo(z) < 0)}

See also the Comparable Javadoc page
Consistency between `compareTo` and `equals`

It is also strongly recommended that that `compareTo` and `equals` be consistent

\[
(x \text{. compareTo}(y) == 0) == (x \text{. equals}(y))
\]

That is: `compareTo` should say two objects are equal if and only if `equals` says that they are equal

So we should always have one of the scenarios below

- But it is *not* enforced by the compiler
- Up to the programmer to stay consistent

Both equal

\[
\begin{align*}
\text{obj1.equals(obj2)} & \quad == \; true \\
\text{obj1.compareTo(obj2)} & \quad == \; 0
\end{align*}
\]

Both not equal

\[
\begin{align*}
\text{obj1.equals(obj2)} & \quad == \; false \\
\text{obj1.compareTo(obj2)} & \quad != \; 0
\end{align*}
\]

An inconsistent `compareTo` and `equals`

```java
public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        Car c = (Car) other;
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }

    public int compareTo(Vehicle other) {
        if (!(other instanceof Car)) {
            return 0;
        }
        Car c = (Car) other;
        if (!this.makeModel.equals(c.makeModel)) {
            return this.makeModel.compareTo(c.makeModel);
        }
        return getMileage() - c.getMileage();
    }
}
```

- So is a `Car` instance equal to instances of other classes, or not?

Are these `compareTo` and `equals` consistent?

```java
public class Car extends Vehicle {
    public boolean equals(Object other) {
        if (!(other instanceof Car)) {
            return false;
        }
        return (this.makeModel.equals(c.makeModel) &&
                getMileage() == c.getMileage());
    }
```

139
final Car c = (Car)other;
return (this.makeModel.equals(c.makeModel) &&
getMileage() == c.getMileage());
}

public int compareTo(Vehicle other) {
if (!(other instanceof Car)) {
    return -1;
}
final Car c = (Car)other;
if (!this.makeModel.equals(c.makeModel)) {
    return this.makeModel.compareTo(c.makeModel);
}
return getMileage() - c.getMileage();
}

• We have ordered Car instances before non-cars
• Is this viable?

More trouble with related classes
The fix on the previous slide
if (!(other instanceof Car)) {
    return -1;
}
works to ensure consistency when comparing Car instances, but also introduces some odd behavior if adopted as is for Truck

```java
public class Truck extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            return -1;
        }
        // ...
    }
}
```

// And executing statements
Car c = new Car("Honda Civic", 214118);
Truck t = new Truck("Ford F-150", 0, 5);
if (c.compareTo(t) < 0) { System.out.println("c less than t"); }
if (t.compareTo(c) < 0) { System.out.println("t less than c"); }

This will print:
c less than t
t less than c

Comparison across related classes
We need to also enforce an ordering across classes

```java
public class Car extends Vehicle { //...
    public int compareTo(Object other) {
        if (!(other instanceof Car)) {
            return -1;
        }
```
public class Van extends Vehicle {
    public int compareTo(Object other) {
        if (!(other instanceof Van)) {
            return 1; // Vans always come last
        }
        // ...
    }
}

public class Truck extends Vehicle {
    public int compareTo(Object other) {
        if (!(other instanceof Truck)) {
            return 1; // Trucks come after Cars
        } else if (other instanceof Van) {
            return -1; // Trucks come before Vans
        } else {
            return -1; // Trucks before anything else
        }
    }
    // ...
}

Comparison across related classes
We need to also enforce an ordering across classes

• Requires subclasses to know about each other
• Not particularly sustainable for a larger number of classes
• Burdensome when adding new subclasses of Vehicle

5.5.2 TODO Find the right home for these three slides

Sorted lists
Once we can compare our objects, we can ensure that our own lists are sorted by changing how objects get added.
Implementation of sorted insert

Sorted insertion is relatively simple, as long as the list is kept sorted

public void add(final Vehicle v) {
    VNode pos = head.next;
    while (pos != tail && v.compareTo(pos.data) >= 0) {
        pos = pos.next;
    }
    addBefore(pos, v);
}

Prevent users from messing up the ordering

public void add(final int idx, final Vehicle v) {
    // Ignore index specified by user
    add(v);
}

Maintaining unique items

Sorted lists can be used to maintain a unique set of items

- The list will hold no duplicates

5.6 Java Collections classes

Collections

So far, we have created our own generic data structures:

- GenericList<T>
- GenericStack<T>
- GenericQueue<T>

These are all examples of collections

- A collection is an object that groups multiple elements into a single unit. It is used to store, retrieve, manipulate, and communicate aggregate data

Java provides this same functionality (and much more) within the java.util package, which include numerous data collection classes that share a common interface
The Collection interface

In Java, the `java.util.Collection<E>` interface specifies common operations that any collection can do

- Add elements
- Remove elements
- Provide an iterator
- Check for membership

```java
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    ...
}
```

The Java collections framework

In addition to Collection, the `java.util` package provides interfaces for several common abstract data types (ADTs)

A collections framework is a unified architecture for representing and manipulating collections

Implementations in the collections framework

In order to make use of these interfaces, we need concrete classes that implement them. The `java.util` package provides several options

```java
java.util.ArrayList<E>
java.util.LinkedList<E>
```

The Collection interface
boolean contains(Object element);
boolean add(E element); // optional
boolean remove(Object element); // optional

Iterator<E> iterator();

boolean containsAll(Collection<?> c);  
boolean addAll(Collection<? extends E> c); // optional
boolean removeAll(Collection<?> c);  
boolean retainAll(Collection<?> c); // optional

void clear(); // optional
Object[] toArray();
<T> T[] toArray(T[] a);

Generics in the Collection interface
Many of the methods in the Collection interface are generic

public interface Collection<E>
    extends Iterable<E> {
    //...
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    
    Iterator<E> iterator();
    
    Object[] toArray();
    <T> T[] toArray(T[] a);

    • E is the type parameter for the collection
    • T is the type parameter for this method
    • The second toArray method checks at run-time that the elements of this collection conform to T
    • Many of these methods are for backwards-compatibility with old versions of Java which did not have generics
      – Sun wanted as much old code as possible to continue to be valid
      – So the only method type changes were those necessary to ensure type integrity

Wildcards and bounds the Collection interface
Other methods in the Collection interface are specified in terms of wildcards

public interface Collection<E> extends Iterable<E> {
    //...
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    //...
}
• \( \exists \) represents any type
  
  – But sometimes this would cause runtime exceptions if it does not properly conform
  
  – So some uses must be bound

• The bound \( \exists \) extends \( E \) guarantees that objects in the given collection conform to the element type \( E \)
  
  – And this can be checked at compile-time

"Optional" methods

Several of the methods in the Collection interface are marked optional

```java
public interface Collection<E> extends Iterable<E> {
  //...
  boolean add(E element); // optional
  boolean remove(Object element); // optional
  boolean addAll(Collection<? extends E> c); // optional
  boolean removeAll(Collection<?> c); // optional
  boolean retainAll(Collection<?> c); // optional
  void clear(); // optional
  //...
}
```

• The "contract" for using java.util collections classes does not require an implementations to support these methods
  
  – Compare with the "contracts" for equals, compareTo, etc.

• However, they must still be implemented
  
  – For example by throwing an UnsupportedOperationException

Lists in Java

In Java:

• The idea of a list and its operations is specified in the List interface

• Concrete implementations of the list abstract data type are provided by several classes, in particular:
  
  – ArrayList
  
  – LinkedList

• Both interface and implementations are generic to allow for arbitrary types of objects to be stored

```java
ArrayList<TYPE> myList = new ArrayList<TYPE>(); // Fill in TYPE
List<TYPE> myList = new ArrayList<TYPE>(); // Fill in TYPE

ArrayList<String> myList1 = new ArrayList<String>();
List<String> myList2 = new ArrayList<String>();
```
The List interface

```java
public interface List<E> extends Collection<E> {
    // Access by index position
    E get(int index);
    E set(int index, E element); // optional
    void add(int index, E element); // optional
    E remove(int index); // optional
    boolean addAll(int index, Collection<? extends E> c); // optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

ArrayList in Java

The List interface and classes support several basic operations:

```java
ArrayList<String> list = new ArrayList<String>();
list.add("Hello"); // contents: {"Hello"}
list.add("World!"); // contents: {"Hello", "World!"}
list.add(1, "Blue"); // contents: {"Hello", "Blue", "World!"}
list.contains("Blue"); // returns true
list.set(1, "Green"); // contents: {"Hello", "Green", "World!"}
```

```java
String temp = list.get(1); // returns "Green"
int curSize = list.size(); // returns 3
```

```java
list.clear(); // contents: {}
curSize = list.size(); // returns 0
```

- Along with selective remove operations
  - But the interface does *not* show us the internal details like (what we called) `capacity`

Implementations of the List interface

To *use* the List interface, we need an actual concrete class that implements it.

```java
interface java.util.List<E>
```

- **Direct access** (`ArrayList`): Stores elements in an underlying array (a contiguous chunk of memory)
- **Linked access** (`LinkedList`): Stores elements in a chain of nodes (not necessarily contiguous)
The **java.util.LinkedList<E>** class

The built-in LinkedList<E> class implements the List<E> and Collection<E> interfaces, and providing additional functionality as well.

<table>
<thead>
<tr>
<th>Interface</th>
<th>java.util.Collection&lt;E&gt;</th>
<th>java.util.List&lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td><code>+ LinkedList() + LinkedList(Collection&lt;? extends E&gt; c)</code> [0.2em]</td>
<td><code>+ ListIterator&lt;E&gt; + ListIterator&lt;E&gt;</code> listIterator(int) [0.2em]</td>
</tr>
<tr>
<td>Query</td>
<td><code>+ E getFirst() + E getLast()</code></td>
<td><code>+ void addFirst(E) + void addLast(E)</code> [0.2em] (plus required interface methods)</td>
</tr>
</tbody>
</table>

The **java.util.ArrayList<E>** class

The built-in ArrayList<E> class also implements the List<E> interface by using an array "under the hood".

<table>
<thead>
<tr>
<th>Interface</th>
<th>java.util.Collection&lt;E&gt;</th>
<th>java.util.List&lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td><code>+ ArrayList() + ArrayList(int) + ArrayList(Collection&lt;? extends E&gt; c)</code></td>
<td><code>+ void removeRange(int,int) + void ensureCapacity(int) + void trimToSize()</code> [0.2em] (plus required interface methods)</td>
</tr>
<tr>
<td>Update</td>
<td><code>+ void addFirst(E) + void addLast(E)</code> [0.2em]</td>
<td></td>
</tr>
</tbody>
</table>

**Iterator and ListIterator**

We have seen the Iterator interface already:

```java
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
    public void remove();
}
```

The ListIterator interface is a subinterface of Iterator that also provides abilities to add/change objects and to move backwards:

```java
public interface ListIterator<T> extends Iterator<T> {
    public boolean hasPrevious();
    public T previous();
    public void add(T x);
    public void set(T newVal);
}
```

### 5.7 Hash codes

Finding elements in a collection **new day?**

We have seen some basic patterns for searching for data in a linear collection (e.g., array, linked list):

- Linear search
- Straightforward
- Worst-case $O(n)$

- Binary search
  - Worst-case $O(\log_2 n)$
  - Requires sorted data and random access

- Can we do better?

**A little bit of magic in our collections**

Suppose we had access to a magic method:
- Given an element as input, it *immediately* returns the location (index) where the element should be located
- We could then go to that location and then determine whether or not the element is actually present
- This would transform search into a *constant-time* lookup!

What sorcery is this!?

*Hash functions!*

- Same idea used in library indexing

**Computing a hash code**

In addition to methods like `toString` and `equals`, every `Object` in Java has its own method for hashing:

```java
public int hashCode()
```

- Converts the *data fields* of an object (its state) to a single integer value

```java
final Integer obj1 = new Integer(2011);
System.out.println(obj1.hashCode());

final String obj2 = new String("2011");
System.out.println(obj2.hashCode());
```

```java
2011
1537246
```

**Overriding the `hashCode` method**

Every class we write *inherits* a default implementation of `hashCode`

- Like the default implementation of `equals`, it is only *guaranteed* to return the same hash code for two objects if they are *identical* (same physical object in memory)

```java
final Person
    p1 = new Person("Tim"),
p2 = new Person("Tim");
final int
    x1 = p1.hashCode(),
x2 = p2.hashCode(); // Not guaranteed to equal x1!
```

But we can override this standard behavior
public class Person {
    private String name;
    //...
    public int hashCode() {
        return 1000 + name.hashCode();
    }
}

Laws for hashCode values
An implementation of hashCode should adhere to these requirements:

• hashCode returns a consistent value every time it is invoked within a single application run (in particular, it must not be random)

• If two objects are equal according to equals, then they must return the same hash code

• If two objects are unequal according to equals, they are not required to return different hash codes

For you as a programmer, this means

• If you override equals, you really should override hashCode
  – But the compiler will not force you to do so

• If you override hashCode, you do not need to override equals

More info on the Javadoc page for Object

Using the magic hash function
With hashCode, every object has its own integer code. This leads to the following idea:

• Store all data in a large array called a hash table

• To store an object, place it at the index matching its hash code

• To look up an object, search the index matching its hash code

Some issues with hashCode
Two objects with the same content should produce the same hash code:

```
Integer obj1 = new Integer(2011);
Integer obj2 = new Integer(2011);
System.out.println(obj1.hashCode() + " : ");
```

For example:

2011 : 2011
Two objects with different content could still produce the same hash code:

```java
String aStr = "Aa";
String bStr = "BB";
System.out.println(aStr.hashCode() + " : " + bStr.hashCode());
```

Conforming output could be:

```
2112 : 2112
```

Some issues with `hashCode`

So we cannot assume that every object has a unique hash code.

- Only so many integers to choose from (in machine representation)
- Two distinct objects collide if they possess the same hash code.
- Hash codes may also be negative

```java
String s = "1999999999999999999999";
System.out.println(s.hashCode());
```

gives:

```
-657850008
```

- We can still make the idea of hash table lookup work, but it requires a bit thought

Resolving collisions in a hash table

Given that collisions are possible, an effective hash table needs to be able to resolve them

- For each “bucket” (cell) in the array, store a list of objects that reside there
  - This is called separate chaining

- So our hash table can be implemented as an array of lists

- Another option for resolving collisions is to store a new entry at the first available cell after its hash index; this idea is known as probing

![Diagram of a hash table with separate chaining](image)
Finalizing the details of the hash table

For each object that needs to be stored in our hash table, we would:

- Compute the hash code
- Convert to an array index
- Go to that array index, and store object in the list at that location

The process for look-up is similar

- Overall running time is bounded by the size of the largest list in the array

Implementing a hash table

```java
final LinkedList<Vehicle>[]
    table = new LinkedList[100]; // Compiler warning!
for (int i=0; i<table.length; ++i) {
    table[i] = new LinkedList<>();
}

// Store an element
Vehicle v = new Car("Honda Civic", 214118);
int code = v.hashCode();
int idx = Math.abs(code) % table.length;
table[idx].add(v);

// Look up
Vehicle t = new Truck("Ford F-150", 1234, 2);
int idx2 = Math.abs(t.hashCode()) % table.length;
if (table[idx2].contains(t)) {
    System.out.println("Truck is in table!");
}
```

The `java.util.HashSet` Class

Rather than reinventing the wheel, we can take advantage of built-in collections for hashing, for example `HashSet`

- Implements the `Collection` interface
- Implements the `Set` interface, so no duplicates allowed
  - `add` returns false if we try to store a duplicate
- The `contains` method is very fast
Using a HashSet

```java
final String s = "If I knew the answer I would tell you that I knew",
// Split on whitespace
final String[] words = s.split(" ");

HashSet<String> hs = new HashSet<String>();
for (String word : words) {
    hs.add(word);
}

System.out.println(" Total words: " + words.length);
System.out.println("Unique words: " + hs.size());
System.out.println(hs);
```

Prints:

```
Total words: 12
Unique words: 9
[that, would, answer, knew, you, tell, the, If, I]
```

Objects get stored in whatever order their hash codes give

Using a HashSet

```java
// Continued from previous
String s1 = "answer";
String s2 = "question";

System.out.println(hs.contains(s1));
System.out.println(hs.contains(s2));
```

true false

- Programmer can *quickly check* if an object is present or not
- Does not need to know how the hash function works
- Saves a lot of time for larger sets (no loops)
The `java.util.HashMap` class

The `java.util.HashMap<K,V>` class implements the `Map<K,V>` interface

- But not the `Collection<E>` interface
- Similar to other collections, we can check size and if it is empty
- Need to use `put` to add a `(key, value)` pair
  - Returns the old value if updated
- Fast look-up via hashing
- *No* iterator access to the map directly; must ask for keys and/or values as a separate collection and iterate
6 Interaction

6.1 Files

6.1.1 Java’s model of files and directories

Files and directories

- A file is a persistent record of data on a computer system.
- On modern operating systems, files live in directories
  - Also called folders

The full “address” of a file is its filepath.

The readme.txt file has the absolute address: /home/user1/readme.txt

Files in Java

File manipulation in Java is done via the java.io.File class

- Deals with names of files and directories (folders)
- Creating a new File(...) object creates something that holds a path name for a file/directory
- Calling createNewFile() or mkdir() will attempt to create an actual empty file/directory

Creating files

```java
public static void makeFile(String filename) {
    final File f = new File(filename);
    if (f.exists()) {
        System.out.println("Exists!");
    } else {
        boolean itWorked = f.createNewFile();
        if (itWorked) {
```

java.io.File

- String fileName
  - «constructor» + File(String)
  - ... «query» + boolean canRead()
  - + boolean canWrite() + boolean
  - isFile() + boolean isDirectory() + boolean exists() + String
  - getName() + int length()
  - ... «update» + boolean
  - createNewFile() + boolean
  - delete() + boolean mkdir() ...

154
System.out.println("Created");
}

• Does not compile!

java.lang.Error: Unresolved compilation problem:
Unhandled exception type IOException

• createNewFile throws an IOException, which is checked

1. Creates File object to store name of file
2. Check if file with that name already exists
3. If not, create new file
   • The boolean return value indicates success or failure

I/O (input/output) exceptions
The createNewFile method has signature

public boolean createNewFile() throws IOException {...}

Since IOException is a checked exceptions, it must be handled in any code that could possibly generate one.

• The throws keyword tells the compiler that the exception might happen, and the compiler then requires the programmer to handle this possibility beforehand

• Options for resolving:
  – Surround with try/catch
  – Propagate exception by adding throws clause

Creating files (the right way)

public static void makeFile(String filename) {
    final File f = new File(filename);
    if (f.exists()) {
        System.out.println("Exists! ");
    } else {
        try {
            final boolean itWorked = f.createNewFile();
            if (itWorked) {
                System.out.println("Created.");
            }
        } catch (final IOException e) {
            System.out.println("Failed!");
        }
    }
}

• Now if file creation fails, the exception is caught and handled
Creating files (another right way)

```java
public static void makeFile(String filename) throws IOException {
    final File f = new File(filename);
    if (f.exists()) {
        System.out.println("Exists! ");
    } else {
        final boolean itWorked = f.createNewFile();
        if (itWorked) {
            System.out.println("Created.");
        }
    }
}
```

• Now, any code that calls the makeFile method must deal with the possible exception

Files in directories
Sometimes we want to create/read/write a file that is in a specific directory

```java
try {
    final File dir = new File("output");
    dir.mkdir();

    final String
    filename = dir.getCanonicalPath()
    + File.separator
    + "data.txt";

    final File f = new File(filename);
    f.createNewFile();
} catch (final IOException e) {
    e.printStackTrace();
}
```

• With mkdir, make directory instead of basic file
• Method getCanonicalPath and static variable File.separator can be used to produce a filename that has all the information needed to place the file in the right place on any file system

Reading and writing to files
File objects can create files and directories
• But they are not used for actually reading and writing files
• Streams allow data to move between a program and a file

![Diagram of data flow between program and file](image-url)
They handle things like the differences between how fast programs can run and how slow data can be written to or read from a disk

6.1.2 Files and streams

Input and output streams: some complications

Input and output generally require two (or more) stages

- One for dealing with the file directly
- Other(s) for getting the data to/from the executing program

![Diagram showing input and output streams]

- The use of BufferedReader and BufferedWriter can sometimes be avoided, but doing so can cause slow-down and other problems, particularly for large files
  - Disk access is very slow compared to the CPU!

- For writing text output to files, it is easier to add a third layer

Writing to a text file

```java
public void writeSampleOutput(String filename) {
    try {
        final FileWriter fout = new FileWriter(filename);
        final BufferedWriter buffout = new BufferedWriter(fout);
        final PrintWriter pout = new PrintWriter(buffout);
```
```java
pout.println("This is line 1.");
pout.flush();
pout.close();

} catch (IOException e) {
    System.out.println("Error: " + e);
}
```

1. Create output stream for file
2. Create buffered output stream for program
3. Create convenient write stream for user
4. Write to file
5. Flush the stream
6. Close the stream

### The FileWriter class

The **FileWriter** has a several constructors

- The basic constructor:

  ```java
  public FileWriter(String name)
  throws FileNotFoundException
  
  - Creates a new file stream each time (also a new file if needed)
  - Will overwrite any pre-existing file of the same name
  ```

- The append constructor:

  ```java
  public FileWriter(String name, boolean append)
  throws FileNotFoundException
  
  append = false  Same as basic constructor
  append = true   Opens any existing file in **append** mode
  
  - Any new output is added onto the end of the file
  - Original file contents are **preserved**
  ```

### Appending to a text file

Creating the **FileWriter** in **append** mode, so existing contents of the file are preserved

```java
public void
writeSampleOutput(String filename) {
try {
    final FileWriter fout = new FileWriter(filename, true);
    final BufferedWriter buffout = new BufferedWriter(fout);
    final PrintWriter pout = new PrintWriter(buffout);
```
Reading from a text file

```java
public void readSampleInput(String filename) {
    try {
        final FileReader fin = new FileReader(filename);
        final BufferedReader reader = new BufferedReader(fin);

        String line = reader.readLine();
        while (line != null) {
            System.out.println("Line: "+ line);
            line = reader.readLine();
        }

        reader.close();
    } catch (IOException e) {
        System.out.println("Error: "+ e);
    }
}
```

Steps

1. Create input stream for file
2. Create buffered input stream for program
3. Read from file
4. Close the stream

Why a loop?

- We rarely know how long a file will be
  - And the length can change
- A loop to read until the end of the file is useful
Reading from a file using BufferedReader

There are different ways to read data using the BufferedReader

- One line at a time

  public String readLine() throws IOException
  
  - Each line read separately (up to but not including line break)
  - Returns String result, or null if the end of the file is reached

- One character at a time

  public int read() throws IOException
  
  - Returns a single character as an int
    * Can cast to char if needed
  - Returns -1 upon reaching end of file

Reading one character at a time

public void
  readSampleInput(String filename) {
    try {
      final FileReader
        fin = new FileReader(filename);
      final BufferedReader
        reader = new BufferedReader(fin);

      int letter = reader.read();
      while (letter != -1) {
        System.out.print((char) letter);
        letter = reader.read();
      }
    reader.close();
  } catch (IOException e) {
    System.out.println("Error: " + e);
  }
}

• Reading one character at a time

• Note casting int to char
  - So print will show that character, not a number
  - (Try it without the cast)

Parsing input with Scanner

The Scanner class (java.util.Scanner) can read input from a variety of sources

- You’ve probably seen this class since the beginning of CS120!

  final Scanner
    scanner = new Scanner(System.in);
• Used primarily for parsing tokens from a stream
• Can check for additional tokens to ensure safe reading
• Can also read from files — more on this shortly

```java
java.util.Scanner

«constructor» +
Scanner(InputStream)
+ Scanner(String) +
Scanner(File) ... «query»
+ String next() + double
  nextDouble() + int nextInt()
+ boolean hasNext() + boolean
  hasNextDouble() + boolean
  hasNextInt() ...
```

Reading files with Scanner

```java
final File inFile = new File("input.txt");
try {
    final Scanner scan = new Scanner(inFile);
    while (scan.hasNextLine()) {
        final String line = scan.nextLine();
        System.out.println("Line: " + line);
    }
    scan.close();
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e);
}
```

If this is so simple, why don’t we always use Scanner?

• BufferedReader has a larger buffer
  – But pay attention to changes in Java versions
  – Generally a better choice for reading line by line
• Scanner is better if you need to parse the input in some way

Files: the big picture

A file is a persistent record of data on a computer system
• All data on a computer is ultimately stored as a series of bits, or binary digits (1s and 0s)

File viewed as text

This is a test.
Hello, World!
Good afternoon, class.

File viewed as bits

Raw bits, grouped into bytes (8-bit blocks):

```
01010100 01101000 01101100 01101111 00100000 01010111
00000000 00110110 01101111 00100000 01100011 01100101
01101100 01101111 01110100 00010000 00000000 00100000
00110100 01110010 01101111 01101101 00100000 01100001
01101111 01100110 00100000 01001000 01101101 01110100
01101101 01101111 01110100 01100100 00100001 00001010
01000111 01101111 01101111 01100100 00100000 01100011
01101100 01100001 01100110 01100110 01101110 01101111
01101111 01101110 00101100 00100000 01100011 01010111
```

The interpretation of these bits is what gives them meaning
Text files

A text file is a computer file that is structured as a sequence of lines of electronic text.

- The bytes in the file are interpreted as human-readable characters (typically with ASCII encoding)
- Lines are delineated by special newline characters

Pros

- Easy to interpret and manipulate for humans
- Fairly robust to data corruption

Cons

- May take up more space than necessary
- Requires conversion to properly deal with numeric quantities

Binary files

A binary file is a computer file whose contents should be interpreted as something other than text.

- Bytes in the file can represent anything
- The interpretation is up to the reader

Some bits

(Same ones as last time)

Interpretations

- We could interpret them as text

  This is a test.
  Hello, World!
  Good afternoon, class.

- We could also interpret them as integers

  1416128883
  543781664
  ...

162
6.1.3 Text and binary files

Comparing text files and binary files
Suppose we want to encode the numbers 123, 1337, and 220 in a file

Write numbers as text

123
1337
220

Convert text to bits:

00110001 00110010 00110011
00001010 00110001 00110011
00110011 00110111 00001010
00110010 00110010 00110000
00001010

Encode integers in binary

00000000 00000000 00000000 01111011
00000000 00000000 00000101 00111001
00000000 00000000 00000000 11011100

What if we interpret these bits as text?

\ldots\ldots9\ldots.

Where the . are unprintable characters

Text or binary?

Recap: numbers 123, 1337, 220
Written as text:

00110001 00110010 00110011 00001010
00110011 00110111 00001010 00110011
00110010 00110010 00110000
00001010

Directly as integers:

00000000 00000000 00000000 01111011
00000000 00000000 00000101 00111001
00000000 00000000 00000000 11011100

- Not a huge difference in sizes here
- Text is easier for humans to process, while binary is easier for machines

Comparing text files and binary files
How about the number 1234567890?

Write numbers as text:

1234567890

Convert text to bits:
Encoding in binary:
01001001 10010110 00000010 11010010

- The binary encoding here is much more compact
- Also much easier for a machine to use for arithmetic operations

**Reading and writing binary files in Java**

The two-step process used for text file I/O also works with binary files

- Just use some different classes for completing the process

```
public void writeSampleOutput(String filename) {
    try {
        final FileOutputStream fs = new FileOutputStream(filename);
        final DataOutputStream dout = new DataOutputStream(fs);
        dout.writeDouble(3.14159);
        dout.writeChar('Z');
        dout.writeInt(101);
        dout.flush();
        dout.close();
    } catch (IOException e) {
        System.out.println("Error: " + e);
    }
}
```

1. Create output streams for file and program
2. Write to the file using a variety of methods (*specific* to the type of data to write)
3. Flush the stream
4. Close the stream
Reading from a binary file

```java
public void readSampleInput(String filename) {
    try {
        final FileInputStream fs = new FileInputStream(filename);
        final DataInputStream din = new DataInputStream(fs);

        final double a = din.readDouble();
        final char b = din.readChar();
        final int c = din.readInt();

        din.close();
    } catch (IOException e) {
        System.out.println("Error: " + e);
    }
}
```

1. Create input streams for file and program
2. Read from the file using a variety of methods (specific to the type of data to read)
3. Close the stream

We must read data by type in the exact order that it was written

- The `read...` methods will read in an exact number of raw bytes
  - Interpret the raw bytes as the given type
- If reading or interpreting fails, the method throws an exception

Complications with binary files

When reading from a file, we either:

- Need to know exactly how much to read from it, or
- Need to use a looping mechanism until we reach the end

With text files, the `read...` methods can return an end of file (EOF) marker if no contents remain in the file:

- With `read`, -1 represents EOF
- With `readLine`, `null` indicates EOF

But this scheme will not work with binary data

- What value would `readInt` return to indicate the end of the file?
- How would we determine if the resulting int
  - Was a valid number from the file?
  - The end of the file?
Reading binary files

For `DataInputStream`, the `readInt` method looks like:

```java
public int readInt() throws IOException, EOFException
```

- An `IOException` is thrown on read failure (e.g., when the file is corrupted)
- An `EOFException` is thrown when a read reaches the end of the file

Because methods for reading binary data have no special value that they can return to indicate the end of the file, they instead throw an `EOFException`. We can use this to detect the end of the file, but it is generally better to find some way to determine exactly how much we should read.

### The ugly way to detect end of file

```java
private int readAllInts(DataInputStream din) throws IOException {
    int count = 0;
    while (true) {
        try {
            din.readInt();
            ++count;
        } catch (EOFException e) {
            return count;
        }
    }
}
```

- Won’t actually loop indefinitely
- Upon end of file, will return count of number of integers in file
- Any read failures resulting in an `IOException` must be dealt with by method caller

### A better solution

Encode additional information at the start of the file (the header) that tells us exactly how much we should read.

- Writing header information plus contents

```java
int size = 10000;
int[] numbers = new int[size];
for (int i = 0; i < size; ++i) {
    dout.writeInt(i);  // Write the file contents
}
```

- Reading header information plus contents

```java
int size = din.readInt();  // Read the header data
for (int i = 0; i < numbers.length; ++i) {
    numbers[i] = din.readInt();  // Read the file contents
}
```
Reading others’ binary files

Why would we read binary files?

• Because other people write them
• Technical standards/specifications describe how the file is arranged
  – First four bytes are a file time, then a forty-byte string identifier, etc
• Or, if we are less lucky, we reverse-engineer the specification from examples

Add examples of closing resources with finally

Including a custom close method as in Marty’s slides

Add examples of try with resources

6.2 GUIs

Events in Java

Events are things like button clicks, mouse moves, and keyboard inputs

• Classes respond to these events by listening for them and handling them

```java
public class DriverWithButtons implements ActionListener {
    private JFrame window;
    private JButton left, right;

    public void actionPerformed(ActionEvent e) {
        // Do the appropriate thing when an action occurs!
    }
}
```

– JButton objects are sources of events
– DriverWithButtons can respond to them when they occur — but won’t necessarily — it needs to be
told when they occur

Handling Java events

The basic Java event model involves three parts:

• Some class of object to create events
  – Like a JButton
• Some class of object to listen for the events
  – Like a DriverWithButtons
• Methods to handle the different events
  – Like actionPerformed(ActionEvent e)
**The JButton class**

JButton objects create an ActionEvent when clicked

- A JButton can be told about each ActionListener interested in the button via the method addActionListener
- When clicked, the JButton will create an ActionEvent and inform the interested listeners by calling their actionPerformed methods
- ActionListener is an interface, so any class we write can implement it!

```java
javax.swing.JButton
+ JButton(String)[0.5em] + int getX()
+ int getY() + int getWidth() + int getHeight() + String getLabel()[0.5em]
+ void addActionListener(ActionListener) + void repaint() + void setBounds(int, int, int, int) + void setSize(int, int)
+ void setLocation(int, int) + void setLabel(String)
```

**The ActionEvent class**

Given an ActionEvent object in our actionPerformed method, we can ask the ActionEvent for its source

```java
public class DriverWithButtons implements ActionListener {
    private JFrame window;
    private JButton left, right;
    //...

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == left) {
            // do left action
        } else if (e.getSource() == right) {
            // do right action
        }
    }
}
```

**Listening for events**

An event listener must tell an event originator that it wants to know about events.

```java
public class DriverWithButtons implements ActionListener {
    //...
    public DriverWithButtons(Window window) {
        // initialize window here
        left = new JButton("Left");
        left.setBounds(5, 5, 100, 20);
        window.add(left);
    }
```
right = new JButton("Right");
right.setBounds(150, 5, 100, 20);
window.add(right);

left.addActionListener(this);
right.addActionListener(this);

Another source of `ActionEvent` objects: Timer

- The Timer class also creates them

The Timer constructor:

- Takes input `int` for the `delay` (milliseconds)
- Takes an `ActionListener` to respond to events

When running, Timer will create a new event every `delay` milliseconds, over and over again.

```
javax.swing.Timer
+ Timer(int, ActionListener){0.5em}
+ boolean isRunning() + void start() + void stop()...
```

Using a Timer for animation

The Timer’s repetitive event generation can be used to perform basic animations, one "frame" at a time.

```
public class DriverWithTimer implements ActionListener {
    private JFrame window;
    private Timer timer;
    //...
    public DriverWithTimer() {
        // create window
        timer = new Timer(200, this);
        timer.start();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == timer) {
            doAnimation();
        }
    }
    //...
}
```
Listening for other events

There are more interfaces besides `ActionListener` for handling other sorts of events.

<table>
<thead>
<tr>
<th>implements</th>
<th>Methods</th>
<th>Listener</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ActionListener</code></td>
<td><code>actionPerformed(ActionEvent e)</code></td>
<td><code>addActionListener(...)</code></td>
</tr>
<tr>
<td><code>MouseListener</code></td>
<td><code>mouseClicked(MouseEvent e)</code></td>
<td><code>addMouseListener(...)</code></td>
</tr>
<tr>
<td></td>
<td><code>mouseEntered(MouseEvent e)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>mouseExited(MouseEvent e)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>mousePressed(MouseEvent e)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>mouseReleased(MouseEvent e)</code></td>
<td></td>
</tr>
<tr>
<td><code>KeyListener</code></td>
<td><code>keyPressed(KeyEvent e)</code></td>
<td><code>addKeyListener(...)</code></td>
</tr>
<tr>
<td></td>
<td><code>keyReleased(KeyEvent e)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>keyTyped(KeyEvent e)</code></td>
<td></td>
</tr>
</tbody>
</table>

The `KeyListener` interface

```java
public interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- `keyPressed()` runs as soon as the user pressed a key
- `keyReleased()` runs as soon as the user lets go of a key
- `keyTyped()` runs in between the two others, but only if doing so would generate a valid Unicode printable character.
  - Pressing `A` would cause all three methods to run
  - Pressing the up-arrow would cause only the first two methods to run

The `KeyEvent` class

The two `get...()` methods allow us to find out what key(s) were pressed/typed/released during an event.

```java
public void keyTyped(KeyEvent e) {
    if (e.getKeyChar() == 'a') {
        // do 'a' action
    } else if (e.getKeyChar() == 'A') {
        // do 'A' action
    }
}
```

- For `keyTyped`, we get the char that would be printed when typing
- Allows us to differentiate between events caused by multiple keys (e.g., Shift+A), as `getKeyChar` returns at most one char

The `KeyEvent` class (continued)

- The `getKeyChar()` method returns a meaningful value only if the key that was used is associated with a printable Unicode character.
• The `keyCode()` method works for many other keys on a standard keyboard (e.g., arrow keys, shift, control)

Large number of pre-defined static int values for "virtual" key-codes in the KeyEvent class

```java
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_SPACE) {
        // do 'space' action
    } else if (e.getKeyCode() == KeyEvent.VK_UP) {
        // do 'up' action
    } // ...
}
```

`java.awt.event.KeyEvent` ...
«query» + char getKeyChar() + int getKeyCode()
7 More data structures

7.1 Trees

From lines to planes

• So far, all the ways we’ve seen to organize data have been *linear* — arrays and lists

• But we can devise *arbitrarily sophisticated* systems of linked information

• A *tree* is another linked structure with
  – One *root* node
  – From each node, several *children* and one *parent*
    * Except the root node has no parent
    * Binary trees have up to two children per node
  – *Leaf* nodes have no children

Trees for ordered data

One common use of binary trees is for storing ordered data

```java
public class GenericSet<A extends Comparable<A>> implements Iterator<A> {
    private int size=0;
    private TreeNode root=null;

    private class TreeNode {
        final A datum;
        TreeNode left, right;

        TreeNode(A datum) {
            this.datum = datum;
            this.left = null;
            this.right = null;
        }
    }
}
```

Adding a node

When finding a place for a new datum in our tree

• Do we look to the left subtree?

• Or to the right subtree?

We use the order to decide

• Elements less than a node’s datum go in the left subtree

• Elements greater than a node’s datum go in the right subtree
Adding a node

In `GenericSet`

```java
public boolean add(final A datum) {
    if (root == null) {
        root = new TreeNode(datum);
        size = 1;
        return true;
    } else {
        return root.add(datum);
    }
}
```

- This is complicated!
- Lots of `null` checks
- How can we simplify it?

In `TreeNode`

```java
public boolean add(final A d) {
    final int rel = datum.compareTo(d);
    if (rel == 0) {
        return false;
    } else if (rel < 0) {
        if (left == null) {
            left = new TreeNode(datum);
            size += 1;
            return true;
        } else {
            return left.add(datum);
        }
    } else {
        if (right == null) {
            right = new TreeNode(datum);
            size += 1;
            return true;
        } else {
            return right.add(datum);
        }
    }
}
```

Another approach

In `GenericSet`

```java
public abstract class GenericSet<A extends Comparable<A>> implements Iterator<A> {
    private int size=0;
    private TreeNode root=new TreeNode();
```
public GenericSet() { }

public boolean add(final A datum) {
    return root.add(datum);
}

• This is an example of the Null Object pattern:
  – Replace (some) uses of null with an object
  – Simplify code with fewer null checks
  – Reduce occasions of null pointer exceptions

In TreeNode

private class TreeNode {
    A datum=null;
    boolean empty=true;
    TreeNode left=null, right=null;

    public boolean add(final A d) {
        if (empty) {
            size += 1;
            empty = false;
            datum = d;
            left = new TreeNode();
            right = new TreeNode();
            return true;
        } else {
            switch (Integer.signum(datum.compareTo(d))) {
                case 1:
                    return left.add(d);
                case -1:
                    return right.add(d);
                default:
                    return false;
            }
        }
    }
}

Finding values
How can we determine whether our tree contains some value?

In TreeNode

public boolean contains(A d) {
    if (empty) {
        return false;
    } else {
        switch (Integer.signum(datum.compareTo(d))) {
            case 1:
                return left.contains(d);
            case -1:
                return right.contains(d);
        }
    }
}
In **GenericSet**

```java
public boolean contains(final A datum) {
    return root.contains(datum);
}
```

Finding nodes

Instead of just a boolean, it can be helpful to return the tree node containing a value

In **TreeNode**

```java
public TreeNode getNode(A d) {
    if (empty) {
        return this;
    } else {
        switch (Integer.signum(datum.compareTo(d))) {
            case 1:
                return left.getNode(d);
            case -1:
                return right.getNode(d);
            default:
                return this;
        }
    }
}
```

In **GenericSet**

```java
public boolean contains(final A datum) {
    return !root.getNode(datum).empty;
}
```

Using **getNode**

We can use **getNode** to simplify **add** as well

```java
public boolean add(final A datum) {
    final TreeNode location = root.getNode(datum);
    if (location.empty) {
        size += 1;
        location.empty = false;
        location.datum = d;
        location.left = new TreeNode();
        location.right = new TreeNode();
        return true;
    } else {
        return false;
    }
}
```
Removing values

How do we remove values from a tree-set?

- If the value is at a leaf node, it’s easy

```java
// In TreeNode
public boolean isLeaf() {
    return !empty && left.empty && right.empty;
}
```

We can simply set that node to be empty, and null out its left- and right-subtrees

- But if it’s a branch, we have two subtrees — how do we combine them?
  - One idea: take either the maximum value of the left subtree, or the minimum value of the right subtree, to be the new value at this node

Maximum and minimum value nodes

How can we find the node with maximum or minimum value in a tree?

```java
public TreeNode getMinNode() {
    if (left.empty) {
        return this;
    } else {
        return left.getMinNode();
    }
}
```

```java
public TreeNode getMaxNode() {
    if (right.empty) {
        return this;
    } else {
        return right.getMaxNode();
    }
}
```

Maximum and minimum value methods become straightforward

```java
public A getMin() {
    return root.getMinNode().datum;
}
```

```java
public A getMax() {
    return root.getMaxNode().datum;
}
```

A remove method for GenericSet

The first part of removing a value is just finding the node containing that value, and then removing it from the tree.

- For the moment, we’ll imagine a TreeNode method `remove` as well, which we implement next

```java
public boolean remove(final A datum) {
    final TreeNode location = root.getNode(datum);
    if (location.empty) {
        return false;
    } else {
        location.remove();
    }
}
```
A remove method for TreeNode

We can identify three cases:

- The node we need to remove is at a leaf
- The node we need to remove is at a branch, and we can find (let’s say) the maximum of the left subtree
- Is there another case?
- The node we need to remove is at a branch, and its left subtree is empty

```java
public void remove() {
    if (isLeaf()) {
        empty = true;
        datum = null;
        left = null;
        right = null;
    } else if (left.empty) {
        empty = right.empty;
        datum = right.datum;
        left = right.left;
        right = right.right;
    } else {
        final TreeNode maxLeft = left.getMaxNode();
        datum = maxLeft.datum;
        maxLeft.remove();
    }
}
```

More binary tree ideas

There are many other features we could consider for binary trees

- **Keep it balanced** — with a minimum possible depth
  - Ensures that search is $O(log \, n)$, at the cost of maintaining balance
  - Further variations which keep the tree “approximately” balanced, but at lower cost than true balance

- **Splay** trees keep recent additions at/near the root for fastest retrieval

- **2-3 trees** allow
  - Some branches to have 1 value and 2 children
  - Other branches to have 2 values and 3 children

- **B-trees** generalize 2-3 trees so that each branch can have $n$ values and $n+1$ children

- And many more!
7.2 Graphs

Trees as models

- Binary trees and their variations are useful for storing and retrieving data
- Trees are also useful for *modeling* situations which are external to the computer
- For examples, each node may represent a possible state of some system, and its children are the result of possible actions we might take
- The children of a branch can be *labelled* with the action giving rise to each child
- Very often, these trees are *not* binary

Game trees

Trees are common representations for analyzing game play

- The root node represents the current state of play
- The children of the root node represent the results of the current player’s possible moves
- The children of those children represent the results of the next player’s possible moves
- And so on

How big can the tree be?

- Some games are small, and the whole tree could fit in memory
  - For example, tic-tac-toe
- Most games do not have finite trees
- Most games’ trees would branch out very quickly
- A common approach: Combine search with *scoring* intermediate positions
  - Maybe to look a certain number of steps ahead, and then use the score to value positions
  - Maybe to limit the number of branches we search more deeply

A different search problem

*Given:* A maze with a starting point and a goal

*Find:* A path from the start to the goal
Searching through a maze

The general idea for searching is the following:

1. Explore the starting tile:
   • Mark it as visited
   • Add its adjacent tiles to a list of unexplored tiles

2. While we haven’t reached the goal:
   • Select an unexplored tile from the list
   • Mark it as visited
   • Add its adjacent tiles to the list

We need to keep track of a few things:

• What we have already explored/visited
• What we have identified but not yet explored (use a list)

Maze search algorithm

Procedure graphTraversal(M)  // Input is a maze
- L = empty list  // Use either a stack or a queue
- Add s to L
- While L is not empty
  - Remove a tile t from L  // Using pop, dequeue, etc.
  - If t has not been visited
    - Mark t as visited  // Draw a mark on the maze
    - If t is the goal
      - Return success  // Found a path to the goal!
    EndIf
  - Explore t as follows:
    - For each tile t’ adjacent to t
      - If t’ is not a wall
        - Add $t’$ to $L$
    EndIf
  End For
End If  // Otherwise ignore t and proceed
End While
- Return failure  // No path found
End Procedure

Graphs

• The maze is a sort of graph
• A more general linked structure
• Nodes may be connected in (more) arbitrary ways
• We often refer to edges and vertices
Graph search algorithm

The maze search was a specific form of graph search

- Finding a node, and a path to that node

Procedure graphSearch(G, s, t) // Find path from s to t
  - Create list of unexplored nodes L
  - Add s to L
  - While L is not empty
    - Remove first node v from L
    - If {visited(v) is false}
      - Set visited(v) to true // Prevents infinite loops
      - If v == t // Target has been found!
        - Return success // Providing path itself is more complex
      - End If // Otherwise ignore t and proceed
    - End For
  - Return failure // Unable to reach target from start
End Procedure

Types of search

The graph search algorithm uses a list to store unexplored nodes. The behavior of the algorithm depends on how this list is implemented

- **Depth-first search** begins at the root (start) and explores as far down a path as it can before backtracking.
  - Implemented using a LIFO structure (like a stack)

- **Breadth-first search** begins at the root (start) and explores all neighbors at a level before moving down to the next level.
  - Implemented using a FIFO structure (like a queue)

Breadth-first search

Using a list with FIFO access yields breadth-first search:

- As each neighbor of a node gets added, it goes to the **end** of the queue
- Removing the next node to search selects the unexplored node that is **closest** to the starting point
- The search process identifies all nodes reachable in 1 step from the start, then all nodes reachable in two steps, then all nodes reachable in three steps, and so on
- Will produce/identify a shortest path:
  - When we see the target node t, we have identified it using the least number of steps possible from s

Depth-first search

Using a list with LIFO access yields depth-first search:

- As each neighbor of a node gets added, it goes to the **top** of the stack
- Removing the next node to search selects the unexplored node that was **most recently** added
- The search process identifies all nodes reachable in any number of steps along a single path from the start, then all nodes along a second possible path, then all nodes along a third possible path, and so on
• Will find a path from $s$ to $t$ (if it exists):
  – Path is not necessarily the shortest one, though

Visualizing search strategies

How can we store graphs?

• Directly, as node with a list of links
• As a two-dimensional array: adjacency matrix
  – The rows and columns index graph nodes
  – The matrix contains 0 if two nodes are not linked, 1 if they are
• As a links list
  – A list of objects, each of which references two nodes

Variations and notions of graphs

• Graph edges can be directed or undirected
  – A mixed graph contains both
• Edges and/or vertices can be labeled, associated with particular information
• Some graphs are acyclic
• Graphs can be connected or disconnected
  – So a tree is a connected acyclic graph
• Multigraphs can distinguish multiple edges between the same vertices
• A hypergraph contains hyperedges which connect more than two vertices
• And many more!
A  Closing lecture: The road ahead

Roadmap

On the roadmap: programming skills, from basic to solid

On the roadmap: software engineering
On the roadmap: hardware and systems

On the roadmap: theory and numerical techniques
On the roadmap: large applications

CS120 — Foundations of Information Security

CS220 — Compiler Construction

CS340 — Machine Learning

CS452 — Artificial Intelligence

CS454 — Graphics and Modeling

CS476 — Data Visualization

On the roadmap: web technologies

CS120 — Intro. Web Design

CS220 — Web App. Development

CS340 — Mobile Application Development

CS364 — Databases

CS471 — Data Communication
B Course policies (Syllabus elements)

B.1 Textbooks and references

There is no required textbook for this class. However, students often benefit from having a reference at hand, and from having a source of extra study problems. Some texts which you should consider:

- There is a version of the online Programming in Java text from zyBooks for this class section. You can subscribe to this book, which will give you electronic access through the end of the semester, and the ability to make a print copy as well if you like. You may be eligible for a discount this semester if you subscribed to this book in a previous semester.

  To subscribe to this textbook:
  1. Sign up at zyBooks.com
  2. Enter zyBook code UWLAXCS220MaraistFall2019
  3. Click Subscribe


- O’Reilly books are often good quality references, although they tend not to have exercises.

B.2 Email and web page

The course website on the cs.uwlax.edu domain will be the primary means of communicating reference information across the whole class; electronic mail will be our primary means of personal communication and certain announcements.

Course web site. The main web page for this class is listed at the beginning of this document. All class announcements will be posted to that page, and you are responsible for checking it regularly. That page also includes an RSS feed for updates. There are several services which will provide email updates from RSS feeds which you can find by a web search; if you choose to use one, pay attention to how often they check the feed and send email.

Email. I will expect you to check your email regularly, and to read and understand messages relevant to this class. In particular, my feedback on your work will be delivered by email. By default I will use your school email address which I receive as part of the information about you that the university gives me, but I am happy to also use a different email address if you email it to me from your school email address. It is your responsibility to make sure that I have an email address which you can and will access regularly, and which you check at least once per business day. Note that we will not use electronic mail for submitting assignments; see the Submission and assessment of assignments section below. My university email account is the only forum which I regularly check; you should not attempt to communicate with me for class business via other email addresses, in-Canvas messaging, or other forms of social media.

For assistance with email or other matters relating to university computer and network services, contact ITS by phone at 608/785-8774, in person on the first floor of Wing Technology Center, or by email to helpdesk@uwlax.edu.

In general, during the semester I will respond to emails with questions about the material, requests for appointments, and other time-sensitive matters within one business day. For administrative matters, requests for regrades, or other matters which can wait a short while, I will usually respond within a week.

When you use email, make sure that you:

Include your full name. There’s a small army of you, and one of me. Make sure it is easy for me to know who you are.

Mention this class by name or number. All of your instructors are almost certainly teaching more than one section.

Write professionally. Observe the forms of casual business writing, write in complete sentences, and use your spellchecker. Keep in mind that email to an instructor about a class is a different medium, and requires a different voice, than texts to a friend.

I have posted links to a number of guides to effective emailing on a web page of resources accessible from my University home page given above.
B.3 Attendance

I expect you to attend class. Our class meetings will be the only source for some class material, and will be the only venue for in-lab assignment components and tests. There are no "makeups" for in-class participation opportunities. If you miss class, it is your responsibility to get notes from a classmate. We will not use class time, nor prioritize office hours and appointment times, to review things missed due to nonattendance. When I keep attendance records for a class, this record will reflect attendance for all, or essentially all, of the class period.

Final examination times are scheduled by the university; make sure to plan any end-of-semester travel around them. Should an exam need to be rescheduled according to the university's limit on the number of exams a student may take on the same day, you must give me notice as soon as you become aware of this situation. I will normally reschedule your exam to the first exam slot before our normal class slot in which you are not taking and I am not giving another exam.

Admission of latecomers to an examination may be refused after any student completes the exam and leaves the exam room.

I do not expect there to be review sessions for this class outside of regular lecture/lab times.

B.4 Submission and assessment of assignments

Each assignment is to be submitted via the electronic submission system detailed in that assignment. I expect that we will primarily use Autolab in this course, but you must always check each assignment for the correct procedure. We will not be using email for assignment submission; assignments emailed to me will not be considered validly or on-time submitted unless either the particular assignment specifically calls for email submission, or I have specifically instructed you to email me an assignment. You are responsible for ensuring that you upload the correct file to Autolab, and in the case of multifile submissions packed as a ZIP archive, for ensuring that the correct contents are all included in the submission: in particular, make sure that Autolab’s output reports that it found correctly-packed work; or if we use Canvas or some other submission mechanism, that you re-download your work to make sure that the server actually has the file(s) you expect. Submissions for programming assignments should consist of fully-functional code which behave as specified in the assignment.

The deadlines for the different types of assignment are as follows (except where a particular assignment specifies otherwise):

- Projects and homework specified on the course site or in slides/notes and turned in online are due by 8:00am of the deadline day.
- In-lab work is to be completed in the lab class for which it is assigned, and is due promptly by the end of the lab for which it is assigned.

My assessment of your coursework will be returned in compliance with FERPA regulations, either directly to you or via email. As described under Email above, I will email you either at your official university email (which only you are authorized to access), or to an alternate email address which you designate. In this way only you will have access to your grades unless you take specific action otherwise.

After you have completed the course, copies or records of your graded material that I retain will be accessible up to 7 weeks into the next academic term (either Spring after Fall or J-term; or Fall after Spring or Summer).

I plan to provide feedback on formative assessments submitted on-time within 21 days of the final deadline for that assessment, and to notify you when circumstances require delay.

B.5 Assignments submitted late

No credit will be awarded for homework or lab work submitted late.

Late credit may be allowed for the final submission of projects:

- Late submissions will be accepted from the deadline up to the time when I download your work from the server for further grading.
- When I download work from the server for grading, that submission point will be closed, and absolutely no further late work will be accepted.
• Do not email late (or otherwise) work to me unless I specifically instruct you to do so. If the Autolab submission point remains open, you may submit late work; if the Autolab submission point has closed, then you may no longer submit late work.

• I will always grade work as soon as I possibly can. Therefore, you should never assume that work will be accepted late at all.

• I will impose a reduction of 10% of the awarded percentage score on the grade of work submitted within 24 hours after the time at which it was due. I will impose a reduction of 40% of the awarded percentage score on the grade of work submitted over 24 hours past the deadline time.

See the Accommodations for individual circumstances section below for extenuating circumstances that impact your ability to meet deadlines or participate in class activities.

B.6 Equity of course execution

This course will be delivered and assessed fairly, in the specific sense that all students in this section will have equivalent opportunities to demonstrate their mastery of the subject, and will be assessed according to the same criteria. The only assessed work and the only criteria for assessing that work, and thus for the grades derived from it, will be as set forth in this syllabus and in the specifications of assignments.

Mindfully attempting to be assessed by more lenient criteria than one's colleagues, or by criteria other than the work for and conduct in this class as described in this syllabus, is unprofessional and will be considered a form of academic misconduct.

B.7 Errors and regrading

If you find an error in the evaluation of your work, you have the right to ask for it to be regraded.

• All requests for regrading must be by email.

• All requests for regrading must detail specifically where the suspected error was made, and what the error is.

• All requests for regrading should be made no sooner than 24 hours, but within one week, of the evaluation of the work being returned to you. If the assessment of some piece of work is returned in stages, the deadline for requesting a regrade will be within a calendar week of when the report containing the suspected error is returned to you.

• To ensure that a uniform standard is applied across the class, all regrading will use the same criteria and rubric applied to everyone else.

• In general, an entire assignment or exam may be regraded in response to a regrading request, even if your request addressed only a proper subset of the original. So make sure that errors to your detriment outweigh errors in your favor.

You will always be notified of errors I find in the evaluation of your work after it is returned to you, as well as any resulting change to your grade, even if you did not request a regrade.

B.8 Collaboration

I encourage you to work together to understand course material. Learning together is a great way to learn and share ideas, and is a useful professional skill. However, in order to actually learn something, it is important that you complete the real work of programming on your own. It is acceptable to:

• Discuss the general approach to an assessed problem with each other.

• Discuss and solve other, unassessed problems together.

• Work together to install software we’ll use, or get it to work properly on individual computers.
Help each other figure out syntax errors when code isn’t compiling.

Help each other isolate and debug problem spots when code isn’t running correctly.

However:

• It is not OK to write code together, or to copy code from anyone inside or outside of the class.

• It is not OK to simply copy code, whether from online, a book or printed article, other people, or any other source. You can use online references to get additional explanations of how Java works, or to learn programming techniques. But the only way to actually gain the skill of programming is to write code yourself.

Any improper behavior with respect to these guidelines will be dealt with as academic misconduct according to University policy.

B.9 Academic integrity and acceptable use policies

Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. I expect you to submit your own original work and participate in the course with integrity and high standards of academic honesty. When appropriate, cite original sources. Plagiarism or cheating in any form may result in a diminished grade or failure of the assignment or of the entire course, and may include harsher sanctions. As necessary I will use resources provided by the university or other services to verify the originality of submitted work. Refer to the Student Handbook for a detailed definition of academic misconduct.

In general,

• You can share ideas, but you may never share code.

• You must independently write all of the code you submit and never copy code from anyone inside or outside of the course to complete an assignment.

• You are expected to be able to fully explain every line of Java code that you write, and may be asked to do so for any given assignment.

In interpreting these general guidelines, "you" should be taken to mean the unit designated to complete one assignment. Except where explicitly stated otherwise in an assignment, all assignments are individual assignments, and it is individuals who may not collaborate on code. Where an assignment is explicitly deemed to be a group assignment, the individuals within a group may freely share material with each other, but never with individuals in other groups.

The article ‘Avoiding Plagiarism’ on the Murphy Library website offers helpful information on avoiding plagiarism. You may also visit the Office of Student Life if you have questions about plagiarism or cheating incidents. Failure to understand what constitutes plagiarism or cheating is not a valid excuse for engaging in academic misconduct. Acadia University offers a light-hearted ten-minute interactive tutorial on avoiding plagiarism at library.acadiau.ca/sites/default/files/library/tutorials/plagiarism

UWL and UWS policy also mandates responsible use of shared computing resources. In particular, your authorization for the use of administrative server resources such as course management systems (like Canvas), program submission and autoevaluation systems (like AutoLab), the course web site, or other assigned systems is strictly limited to the purpose described in course assignments and other material. Any disruption, exploration and/or exfiltration of system components is strictly prohibited, and may also constitute academic misconduct. Login credentials to university and other systems used for coursework may not be shared, and any such sharing may be taken as firm and sufficient evidence of assignment non-originality. More information about the UWS policy on Acceptable Use of Information Technology Resources is available at www.wisconsin.edu/regents/policies/acceptable-use-of-information-technology-resources

B.10 Professional conduct

Interacting with peers and with me in a constructive, respectful and professional manner, being a constructive and supportive presence in class, handling difficulties with grace and resilience, operating as an autonomous and responsible adult, fulfilling commitments, and approaching work with enthusiasm are all valuable professional (and life) skills.
and are firm expectations of this class. Part of your final grade in this class will be determined by the quality and consistency of your professional conduct, whether online, in class, or in office hours.

One aspect of being a constructive and supportive presence in class is simply not being disruptive to the class. Attendance carries the obligation of being a constructive presence, or at least, a non-disruptive presence. In particular:

- Cell phones and other electronics must be silenced for the duration of class. Consider using an app like [Shush!](https://www.shushapp.com) or [Silent Time](https://www.silenttimeapp.com) (for Android), or [AutoSilent](https://apps.apple.com/us/app/autosilent/id1083213476) (for iPhone) to manage silencing your devices automatically.

- If you need to arrive to class late or leave early, be mindful of creating a minimum of disruption: sit near the exit and on the end of the aisle, pack lightly, and avoid using materials in class which are noisy on packing/unpacking.

- Research has shown that screen use in class is distracting not only to the student using a device, but also to that student’s neighbors. So if you plan to use a screened device in class, I’ll expect you to sit in the back row so that your screen distracts the fewest people. Likewise, if you plan not to use a screened device, you should sit away from the rearmost rows.

In cases of egregious, repeated or persistent disruptive conduct, of mindful discourtesy or of any intimidation of anyone in class, or of isolating or shaming conduct based on gender, race or other identity issues, I may require you to leave class immediately, possibly on an ongoing basis.

Findings of academic misconduct and/or unacceptable use of course resources may also result in loss of graded credit for professional conduct. In particular academic misconduct on a project, major assignment or any examination, as well as multiple instances academic misconduct and/or unacceptable use of course resources, will result in the loss of most if not all credit for professional conduct.

In laboratory sessions,

- Be gentle with lab computers.

- Speak in quiet tones in the lab to avoid disturbing others.

- It is permissible to assist neighboring colleagues with debugging when they are stuck on a particular problem. However:
  - You may not copy any aspect of your lab work from a colleague, nor provide your work to them for duplication.
  - You may not interrupt colleagues, who have their own work to do, to repeatedly ask for help; raise your hand and I will answer your question as soon as I can. Moreover you should remain at your own computer unless you are leaving the lab; moving about the lab for conversations is disruptive to others.

- Use of headphones in lab is unprofessional and strongly discouraged. I will frequently announce important material, and cannot repeat it individually simply because you excluded yourself from listening with the group.

- Do not touch computer screens; use the mouse when indicating particular items to me or to a colleague.

- Food is not allowed in the lab. Drinks in closed containers are permitted but may not be placed on the same desk as a computer or keyboard — keep them on the floor, where a spill will not destroy equipment.

- We recommend that you use hand sanitizer when leaving the lab; keyboards and mice are notorious vectors for communicable disease.

### B.11 Concerns or complaints

If you have a concern or a complaint about either the course or me, I encourage you to bring it to my attention. My hope would be that by communicating your concern we would be able to come to a resolution. If you are uncomfortable speaking with me, or if you feel your concern has not been resolved after bringing it to my attention, you can contact my department chair or the Office of Student Life.

I normally give anonymized examinations: you will sit at a desk tagged with your name; rather than writing your name on the exam, you will write the number on that tag. The anonymity allows us all to be more confident in the accuracy and uniformity of assessment across the class. However, that anonymity extends only through the completion of assessing the individual exam questions. After marking I will de-anonymize the exam papers to understand both individual and group trends and weaknesses, and to address them through subsequent improvements to the class. So exam papers should not be considered an anonymous forum for suggestions or complaints.

B.12 Sexual harassment

As an employee of the University of Wisconsin-La Crosse, I am a mandated reporter of sexual harassment and sexual violence (which include sexual assault, domestic violence and stalking) that either takes place on campus or otherwise affects the campus community.

So if I receive detailed or specific information about an incident such as the date, time, location, or identity of the people involved, I am obligated to share this with UWL’s Title IX Coordinator in order to enable the university to take appropriate action to ensure the safety and rights of all involved. It does not matter whether the incident took place on- or off-campus; it matters only that a person who is a member of this campus was involved in the incident.

It is possible that course assignments may lend themselves to disclosure, but you should not share any details of an incident with me until you have discussed your options under the new Title IX guidelines. There are confidential reporters available to students at UWL where you can have this discussion.

For students not wishing to make an official report, there are confidential resources available to provide support and discuss the available options. The contact in Student Life is Ingrid Peterson, Violence Prevention Specialist, 608/785-8062, ipeterson@uwlax.edu. For more resources or to file a report, please see www.uwlax.edu/violence-prevention.

I am also happy to help direct you to counseling and support services. Simply ask me to assist you in locating a confidential reporter and I will help you to do so.

B.13 Class interruptions and cancellations

In the event of a campus incident that impacts the availability of teaching spaces, any changes or cancellations will be communicated to you via your university email account. Depending on the incident, some or all of the information might be posted on the UWL home page.

In the event of inclement weather, we will follow the [University’s closure policy](#). If classes are not canceled, I will make every effort to be in class on time, and so should you. Please do not send me email asking whether class is going to meet; instead, check the university website. The university’s emergency readiness plan is available online, that page also describes sign-ups for individual emergency alerts. In the event of a cancellation, consult the course homepage for any alternative assignments or other arrangements.
B.14 Accommodations for individual circumstances

It is my goal that all students have equivalent opportunities to succeed in this class. This section discusses the general procedures for alternative assessment accommodations in this class, as well as a number of specific situations for which there are standard mechanisms and policies in place to achieve the goal via accommodations for individual circumstances.

General procedures and constraints. Students may propose alternative assessments for assignments and exams for matters outside of a student’s control such as documented non-chronic illness, bereavement, unplanned university equipment unavailability, or university program travel or activities.

- In almost all cases, you will work with a campus office (usually one of the ACCESS Center, Veterans Services Office, or Office of Student Life) to design and manage your accommodations. They will have confidential access to the full details of your situation, and so they will be the sole authority who can certify that the accommodations you propose are both necessary and sufficient for your situation. Moreover all accommodations shall be reviewed by the same office: the necessity and sufficiency of the overall accommodation for your situation cannot be accurately assessed otherwise.

- Any accommodation must also ensure that the required objectives for this course are assessed as thoroughly as under standard procedures. It is my role to judge whether any proposed accommodation meets this requirement.

- It is your responsibility to propose assessment alternatives which which are both approved by the overseeing campus office as necessary and sufficient to accommodate your circumstances, and approved by me as appropriate for the original assessment’s objectives.

- All requests for accommodation must be accompanied by appropriate supporting documentation. In most cases this documentation will be reviewed by a separate group on campus such as the ACCESS Center or Veterans Services Office, and I will not see specific details. Where no such campus group applies, the specific form of documentation will be at my discretion.

- Proposals for alternative assessment must be made at least ten calendar days before any relevant major deadline or exam. If a proposal cannot be made in time due to medical or other emergency, the proposal should be made at the earliest possible point.

- Alternative assessment proposals should address relevant big-picture issues in addition to immediate course matters.

- Alternative assessment proposals must be explicit, and must be sent only by email or in writing.

- Students proposing alternative assessments should never simply assume that their proposal will be granted verbatim, and must allow time for thoughtful review of all proposals.

- Extracurricular and student groups/activities, planned personal trips, and similar elective activities are not considered to be outside of a student’s control, and do not qualify for alternative assessment.

- Accommodations are generally not available for the activities of other classes. Do not schedule activities for other classes during the lecture/lab/exam times of this class; you are not “free” at those times.

- Accommodations should enable you to complete the assessments for this class during the regular semester. I will avoid recording incomplete grades as part of an alternative assessment plan for any situation which has previously been addressed by accommodation, whether at UWL or other institution, whether via the ACCESS Center or not. Incomplete grades will also not be used where an advisor’s or other credible recommendation for a reduced load, for a particular semester or on an ongoing basis, was disregarded or avoided; you are expected to design a feasible schedule with your (formal and informal) advisors.

- The goal of providing equivalent opportunities to succeed in this class to all students enrolled in the class means that there will not be individual variations to assessment in this class except as allowed in this section. Thus “extra credit” and other alternative assessments not included in the class-wide assessment plan are specifically disallowed.
Disabilities and medical conditions. Accommodations for a documented disability or medical condition are made via the ACCESS Center. You must contact The ACCESS Center and meet with an advisor to register documentation of your situation, and to develop and propose alternative assessments.

- Examples of the disabilities and conditions for which this procedure applies include, but are not limited to: ADHD; autism spectrum disorder; acquired brain injury; PTSD; and physical, sensory, psychological, or learning disabilities.

- The ACCESS Center is located at 165 Murphy Library, and is reachable by phone at 608/785-6900 and by email at ACCESSCenter@uwlax.edu.

Interactions with the ACCESS Center and with instructors should be initiated promptly. For issues and conditions identified prior to the semester, you should contact the ACCESS Center prior to the semester in order to propose and confirm an accommodation plan before assignments are due. For issues arising during the semester, you should contact the ACCESS center to initiate their accommodations process promptly after a diagnosis. Accommodations will not be applied retroactively in the case of a delay in initiating the ACCESS Center process. Once some alternative assessment accommodation is arranged for you via the ACCESS Center in this class, any other accommodations for you as well as any changes or extensions to your accommodations, including those arising from changes in your underlying condition or disability, must also be arranged via ACCESS Center procedures (see Changes to accommodations below), and must follow the procedures described elsewhere in this syllabus.

You can find out more about services available to you with disabilities at The ACCESS Center website, www.uwlax.edu/access-center.

Veterans and active military personnel. Veterans and active military personnel with special circumstances (e.g., upcoming deployments, drill requirements, disabilities) are welcome and encouraged to discuss these issues with me, and I expect you to do so as far in advance as possible.

For additional information and assistance, contact the Veterans Services Office, www.uwlax.edu/veteran-services. Students who need to withdraw from class or from the university due to military orders should familiarize themselves with the university’s current military duty withdrawal policy, catalog.uwlax.edu/undergraduate/academicpolicies/withdrawal.

Religious accommodations. Per the UWL Undergraduate and Graduate Catalogs, “any student with a conflict between an academic requirement and any religious observance must be given an alternative means of meeting the academic requirement. The student must notify the instructor within the first three weeks of class of specific days/dates for which the student will request an accommodation. Instructors may schedule a make-up examination or other academic requirement before or after the regularly scheduled examination or other academic requirement.”

University athletics. Student athletes are expected to submit the semester’s full schedule, including expected travel times and possible championship tournaments, by the end of the first week of class. I realize that your coaches’ official letter may not be ready by that time: that letter can come later. But you are able and expected to collect and convey the information yourself, and later follow up with the official documentation.

In the event of cancellations or postponed events, I expect you to inform me in email before our next class meeting of the cancellation. In that email, you should also indicate to the best of your knowledge whether the university is attempting to reschedule the event later in the semester.

Changes to accommodations. Accommodations can change by mutual consent to reflect changed circumstances. Changes should follow the same review and implementation mechanism as the original accommodation; in particular where the ACCESS Center reviewed and recommended original accommodations, I will expect changes or parallel accommodations to be reviewed and recommended through the ACCESS Center.
C Departmental learning outcomes for CS120 and CS220

The UWL Department of Computer Science publishes student learning outcomes (SLOs) for all classes to ensure consistency across classes and sections. This section reproduces the departmental SLOs for CS120 and CS220. These outcomes were last updated in Spring 2019.

C.1 Student learning outcomes for CS120

Students shall be able to:

• Write Java programs using non-parallel control instructions, including assignment, method call (void and non-void), if, while, for (iterative).

• Write and evaluate expressions using literals, variables, parentheses, and the following operators (note that the precedence and associativity of the operators is included):
  – numeric: − (negation), +, −, /, *, %, ++ (postfix), − (postfix), =, !=, <, <=, >, >=
  – boolean: !, &&, ||
  – String: +
  – Object: instanceof

• Write and evaluate primitive expressions involving mixed types, widening, and casts.

• Write and evaluate variable declarations (including final variables), that demonstrate an understanding of local, private, public and protected scope; and the use of the this notation.

• Write code that demonstrates an understanding of the principle of information hiding by choosing correct scope.

• Compose and evaluate code that demonstrates and understanding of object binding, the null notation and orphan objects.

• Draw, interpret, and trace code using object diagrams.

• Draw and interpret class diagrams; including scope annotations, and aggregation and inheritance relations.

• Develop programs involving all of the following algorithm patterns: variable content swap, cascading if instructions, counting loops, linear search, selection sort, object access shared by multiple classes, method callback.

• Write and evaluate code that uses inheritance, constructor overloading, method overriding, and uses the super notation to invoke a superclass constructor.

• Demonstrate an understanding of method preconditions and postconditions using informal logical descriptions.

• Identify and correct code exhibiting infinite loops, NullPointerExceptions and ArrayIndexOutOfBoundExceptions.

• Debug by inserting println instructions.

• Adhere to fundamental programming style conventions, including using meaningful identifiers, intelligent inclusion of comments and proper indentation patterns.

• Write and evaluate code involving one-dimensional arrays.

• Write and evaluate code with import declarations.

• Write and evaluate code involving the following standard Java classes, methods and constants:
  – Object: equals, toString
- String: length, charAt, toUpperCase, toLowerCase, substring (both versions), indexOf
- Math: random, abs, sqrt, trigonometric functions, pow, PI and E
- Scanner (using System.in): nextX, hasNextX
- System.out.print and System.out.println
- GUI:
  * JFrame
  * Container: add, remove, repaint
  * JComponent: paint
  * JButton
  * JTextField

• Write and evaluate code involving event handling with JButton and JTextField objects.

C.2 Student learning outcomes for CS220

Students shall be able to:

• Use a production quality IDE to write, debug, refactor (via renaming), and execute programs.

• Write well-designed code involving inheritance, overloading, and overriding.

• Write and evaluate code involving multi-dimensional arrays.

• Evaluate code that demonstrates the internal one-dimensional structure of multidimensional arrays.

• Write code that demonstrates an understanding of the separation of abstraction and implementation: making correct choices between alternative linear container representations.

• Write and evaluate preconditions, postconditions and class invariants using informal notation.

• Write and evaluate code involving exceptions, try blocks, throw instructions, throws qualifiers, and finally blocks.

• Understand the relationship between files and directories in a hierarchical file directory system and name files with both relative and absolute names.

• Understand the distinction between binary and text files, select between them, and translate data of each type to the equivalent other type.

• Read and write code using the following classes and associated methods:
  – File: delete, exists, getName, isDirectory, isFile
  – DataInputStream: close, read, write
  – DataOutputStream: flush, close, read, write
  – BufferedReader: close, read, readLine
  – PrintWriter: close, print, println
  – Scanner (with input streams): next, hasNext

• Evaluate recursive definitions.

• Write and evaluate recursive methods; both void and non-void.

• Write and evaluate code that uses interfaces and abstract classes.

• Trace the behavior of code using static, dynamic and automatic memory, understanding the usage of each.
• Write and evaluate code implementing singly and doubly linked lists including traversal, item insertion, item removal, and uses both non-sentinel and sentinel cells.

• Write and evaluate code that uses inner private classes after the style of java.util.LinkedList.

• Write and evaluate code implementing stacks and queues.

• Write and evaluate code that uses the for-each statement.

• Read and write code involving the following Java classes.
  
  – java.lang.Comparable
  – java.util.Comparator
  – java.util.Collection
  – java.util.List
  – java.util.Iterator

• Perform counting analyses on linear, polynomial and logarithmic algorithms.

• Give the relative ordering of logarithmic, linear, and polynomial-time algorithms.

• Write and evaluate programs involving all of the following algorithms: binary search, linear search (for both arrays and lists), insertion sort, merge sort, and quicksort.

• Interpret and utilize object type conformance and subtype polymorphism.

• Write and evaluate code that uses bounded generic classes.

• Write and evaluate code that uses the wrapper classes including auto-boxing and auto-unboxing.
D  Review exercises

Some problems are marked to reflect particular challenges or interest areas:

Marks problems which use mathematical examples reaching beyond the core math requirements of the CS degree. However, all of these problems can be solved simply by implementing the formulas given with the methods in java.lang.Math, even if all of the mathematical concepts are not entirely clear.

D.1  Expressions and assignment

Exercise D.1.1.  Answer these questions by writing short Java programs

• Does subtraction group to the left, or to the right? That is, when we ask Java to evaluate \(100-50-10\), will it evaluate \((100-50)-10\), or will it evaluate \(100-(50-10)\)?

• Does division group to the left, or to the right?

• Does modulus group to the left, or to the right?

• Does multiplication take precedence over addition, as it does in school algebra? That is, when we ask Java to evaluate \(100+50*10\), will it evaluate \((100+50)*10\), or will it evaluate \(100+(50*10)\)?

□

Exercise D.1.2.  Answer the following questions using the java.lang.Math documentation

• What methods does Java provide for logarithms?

• What methods does Java provide for trigonometry?

• What is the difference between floor and ceil?

• What is the difference between floor and round?

• What do the signum methods do? Why are there two of them?

□

Exercise D.1.3.  Write a Java class TempConverter with a static method toCelsius, which takes a Fahrenheit temperature as a double value, and returns the equivalent Celsius temperature.

□

Exercise D.1.4.  Write static methods \(f_1, f_2\) and so on implementing the following mathematical functions on real numbers (double). Do not use methods from the Math class for these.

1. \(f_1(x) = 2x + 1\)
2. \(f_2(x, y) = x^2 + 2xy + y^2\)
3. \(f_3(u) = u^3 + 2u^2 - 3u + 10\)
4. \(f_4(w) = \frac{w+1}{w-1}\)
5. \(f_5(z) = f_3(z) + f_4(2 + z^2)\)

What happens when we call \(f_4(1)\) from a main method?

□
Exercise D.1.5. Write static methods \( g_1, g_2 \) and so on implementing the following mathematical functions on real numbers (double). Do use methods from the Math class for these.

1. \( g_1(x) = \sqrt{2x^2 + 1} \)
2. \( g_2(x, y) = \log_x y \)
3. \( g_3(w) = |w + 10| \)
4. \( g_4(z) = z^{200} \)

What happens when we call \( f_4(1) \) from a main method?

Exercise D.1.6. Most cereals are made primarily of flour, sugar and high-fructose corn syrup. Write a class CerealMaker with a static method announceComposition. Your method should take three integer arguments, representing (respectively) the number of grams of flour, sugar and high-fructose corn syrup in a standard serving of some particular cereal. Your announceComposition method should print a well-formatted announcement of the total number of grams in a standard serving, repeat the number of grams and the name of each ingredient, and then print the total percentage of the standard serving which is sweetener. Your announceComposition method should not return any result.

Exercise D.1.7. You have probably run across the factorial function in your math classes. It is defined by two rules:

\[
\begin{align*}
0! & = 0 \\
n! & = n \cdot (n - 1)! \quad \text{when } n > 0
\end{align*}
\]

We have not yet learned enough Java to implement a factorial method. But we can get ready for when we implement factorial, by writing methods to test our implementation. Notice the difference with the example above — there, we checked what a method was already doing; now, we are setting expectations for what a method will do. This approach is called test-driven development — we write tests first, so that our goals are clear, and so that we can know when our method is correct.

We stub the factorial method by writing an implementation which we know is wrong, but which will compile and run with our tests. By making our tests compile and run (albeit with incorrect results), when we do develop the factorial method, we can do so without worrying that our test infrastructure is lacking.

```java
public class FactorialTester {
    public long factorial(int n) {
        // TODO --- later we will implement factorial correctly
        return -1;
    }
}
```

So starting from the above class, add a main method which tests factorial on several different values.

Exercise D.1.8. Write a class NumberLengthFinder with static method lengthInDigits which takes a number as its argument, and returns the number of digits it takes to write down that number (in base-10). Your lengthInDigits method should

- Take a single long argument, and
- Return a long result.

(For a hint, see p.229)
Exercise D.1.9. Write a Java class `DoubleIt` with

- A method `getDoubled` which takes as parameters a single `int` value, and which returns the `int` value which is twice the value of the argument.

- A `main` method which makes at least three different test calls to `getDoubled`, and which prints for each the test argument, the expected output from the call to `getDoubled` with that argument, and the result returned by `getDoubled`.

Follow our discipline of software design, and complete this work in four steps in order:

0. Stubs of all methods.
1. Tests in the `main` method.
2. A design — but no implementation — for `getDoubled`.
3. The implementation for `getDoubled`.

Make sure your code compiles and runs at each step (add an empty stub for the `main` method at Step 0).

Exercise D.1.10. A `TwoMult` sequence is a sequence of numbers where each number (after the first two) is the product of the two prior numbers. Set up a Java class `TwoMult` with

- A `void` method `printTwoMult` which takes as parameters the first two numbers of a `TwoMult` sequence, and which when completed will print the next three numbers of the sequence.

- A `main` method which makes at least five different test calls to `printTwoMult`, and which displays for each the test arguments, the expected output from a call to `printTwoMult` with those values, and then calls `printTwoMult`.

Follow our discipline of software design, and complete this work in four steps in order, as in Exercise D.1.9.

Exercise D.1.11. Extend Exercise D.1.9 to add another method `printDoubled`. This method should also take as parameters a single `int` value, but `printDoubled` should print a nice message with the argument and its double, and return no result. So for example, when calling `printDoubled(3)`, your method might print

```
3 doubled is 6.
```

Follow our discipline of software design as usual:

0. First add a stub for `printDoubled` to class `DoubleIt`.
1. Add tests for `printDoubled` to the `main` method. Do not remove the old tests for `getDoubled` — old tests let us make sure we do not break old code when adding new code!
2. A design — but no implementation — for `printDoubled`
3. The implementation for `printDoubled` again, make sure your code compiles and runs at each step.

Exercise D.1.12. Write a class `ClockTime` with a method `getClockTime` which

- Take a single `int` argument representing a number of seconds, and
- Returns a string representing the given length of time as a number of hours, minutes and seconds written with a colon between them in the way we usually write clock times.

For example, the result of calling `getClockTime(5025)` would be `1:23:45`. Follow our discipline of software design as usual (so your class `ClockTime` will contain a `main` method in addition to `getClockTime`).

What happens when you run your program on input `7260`? Is the output what you would write (or expect to read) for a clock time? If not, why not? We will come back later to this program and fix this problem.
Exercise D.1.13. Write a class ParityChecker with a single static method isEven which

- Takes a single int argument
- Returns true exactly when the argument is even, that is, exactly when the argument is divisible by 2 with remainder 0.

Exercise D.1.14. Write a class SquaresChecker with a single static method isSquared which

- Takes a single int argument
- Returns true exactly when the argument is a perfect square, that is, exactly when there is some other integer \( x \) such that the argument is equal to \( x^2 \).

Exercise D.1.15. In Warren Buffet’s 2010 philanthropic pledge, he attributed his wealth to a number of factors, one of which was compound interest. Unlike simple interest calculations which assume a constant principal (original investment) value, when calculating compound interest one adds interest payments for a period back into the principal, and then calculates further interest on the higher base value. To determine the amount that an investment made under compound interest will be worth at the end of the investment, we can use this formula:

\[
A = P \left(1 + \frac{r}{n}\right)^{nt},
\]

where:

- \( A \) is the total value at the end of the investment period,
- \( P \) is the original invested value,
- \( r \) is the interest rate as a decimal (so, 0.05 for 5%),
- \( n \) is the number of times per year that interest is compounded, and
- \( t \) is the (whole) number of years of investment.

Write a class CompoundInterestCalculator with a static method getFinalValue which takes the four arguments \( P, r, n, t \) in that order (with \( P \) and \( r \) as double, \( n \) and \( t \) as int), and returns the total value \( A \) (as a double).

D.2 Control structures

Exercise D.2.1. Write a class Grader with static method getLetterGrade which takes an integer argument representing a percentage grade from 0 to 100, and returns a string representing the corresponding letter grade,

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>95 \leq g &lt; 95</td>
<td>A</td>
</tr>
<tr>
<td>92 \leq g &lt; 92</td>
<td>AB</td>
</tr>
<tr>
<td>86 \leq g &lt; 86</td>
<td>B</td>
</tr>
<tr>
<td>82 \leq g &lt; 82</td>
<td>BC</td>
</tr>
<tr>
<td>73 \leq g &lt; 82</td>
<td>C</td>
</tr>
<tr>
<td>60 \leq g &lt; 73</td>
<td>D</td>
</tr>
<tr>
<td>g &lt; 60</td>
<td>F</td>
</tr>
</tbody>
</table>
As a first step, write a main method with examples and expected grade calculations.

Exercise D.2.2. WidgetCo manufactures several different kinds of widgets for re-sale by various vendors. Based on past relationships, sales targets, and other factors, certain vendors are given discount codes which entitle them to a particular discount on their purchases. Write a class `WidgetCoDiscounts` with a static method `getDiscountedPrice` which takes two arguments, a string discount code and an integer base purchase price, and returns the price which should be charged given the particular discount code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>10%</td>
</tr>
<tr>
<td>R</td>
<td>12%</td>
</tr>
<tr>
<td>T</td>
<td>15%</td>
</tr>
<tr>
<td>M</td>
<td>3%</td>
</tr>
<tr>
<td>E</td>
<td>8%</td>
</tr>
</tbody>
</table>

If the discount code is an empty string or does not match any in the above table, then the method should return the original base purchase price.

As a first step, write a main method with examples and expected price calculations, for example

```java
System.out.println("For code S and purchase $100, expect 90, got " + getDiscountedPrice("S", 100));
```

Remember that we compare strings with `.equals` but compare numbers with `==`.

Exercise D.2.3. Write a class `MonthNamer` with a static method `getMonthName` which

- Takes a single integer corresponding to a month of the year, 1 representing January through 12 representing December, and
- Returns a string for the name of the month.

Exercise D.2.4. Square Deal Credit Union offers a program for first-time home buyers to save on the downpayment required for their loan. The downpayment is calculated according to the following table:

<table>
<thead>
<tr>
<th>Purchase price of home</th>
<th>Downpayment required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under $50,000</td>
<td>4% of price</td>
</tr>
<tr>
<td>$50,000-$124,999</td>
<td>$2,000 plus 8% of price over $50,000</td>
</tr>
<tr>
<td>$125,000-$175,000</td>
<td>$8,000 plus 12% of price over $125,000</td>
</tr>
<tr>
<td>Over $175,000</td>
<td>Not eligible for this program</td>
</tr>
</tbody>
</table>

Write a class `SquareDeal` with static method `getProgramDownpayment` which takes one integer argument representing the home purchase price, and returns an integer representing the required downpayment under this program, or -1 if either the home is not eligible or a negative price is entered. As a first step, write a main method with examples and expected downpayment calculations.

Exercise D.2.5. Write a class `TwoSorter` with static method `printInOrder` which

- Takes two arguments of type `int`,
- Returns nothing, and
- Prints the two numbers in ascending numeric order.

As a first step, write a main method with examples and expected output.
Exercise D.2.6. Write a class `ThreeSorter` with static method `printInOrder` which

- Takes three arguments of type `int`,
- Returns nothing, and
- Prints the three numbers in ascending numeric order.

As a first step, write a `main` method with examples and expected output.

Exercise D.2.7. The Wisconsin Wants Walnuts company buys black walnuts from individuals, and shells them for sale to restaurants. Sellers’ walnuts are loaded into baskets which hold about one pound of unshelled nuts, and WWW pays $0.15 for each full basket of nuts. If the final, partially-filled basket is more than half-full, WWW pays the full $0.15 for that basket; otherwise they pay $0.05 for the partial basket. Write a class `WalnutBuyer` with a static method `getPurchaseOffer` which

- Takes a single `double` representing the number of baskets of black walnuts brought by a seller (so for example, 3.25 represents three full baskets and an additional basket which is one-quarter full), and
- Returns the amount that WWW will pay for those baskets.

As a first step, write a `main` method with examples and expected payments.

Exercise D.2.8. Write a class `ParityChecker` and static method `isOdd` which

- Takes one parameter of type `int`, and
- Returns a result of type `boolean` which is `true` exactly when the argument is odd.

Do not use any of the methods in the `Math` class for your method. As a first step, write a `main` method with examples and expected results.

Exercise D.2.9. Write a class `TripleChecker` with a static method `isTriple` which

- Takes a single integer, and
- Returns `true` when the integer argument is exactly divisible by 3.

Exercise D.2.10. Consider three sticks of length two inches, three inches and six inches. We could not form a triangle with these sticks, because one stick is longer than the other two put together. But if instead the sticks had lengths two inches, three inches and four inches, we could make a triangle from those sticks. Write a class `TriangleLengthsChecker` and static method `isTrianglePossible` which

- Takes three arguments of type `int`,
- Returns a result of type `boolean` which is `true` exactly when sticks of the three lengths could form a triangle.

If one of the lengths is zero or negative, your method should return `false`. As a first step, write a `main` method with examples and expected results.

Exercise D.2.11. Write a class `EvenSquares` and two static methods `isEvenAndSquare` and `isEvenOrSquare` where

- Both methods take one parameter of type `int` and have a result of type `boolean`,
- `isEvenAndSquare` returns `true` when the argument is *both* an even number and a perfect square, and
- `isEvenOrSquare` returns `true` when the argument is *either* an even number or a perfect square.

You are free to use any of the methods in the `Math` class for this exercise. As a first step, write a `main` method with examples and expected results for both methods.
Exercise D.2.12. Write a class LeapYearChecker with a static method isLeapYear which determines whether a year is a leap year. The rules and exceptions for determining whether a year is a leap year are:

- Most years are not leap years
- Unless the year is divisible by 4, in which case it is a leap year
- Unless the year is also divisible by 100, in which case it is not a leap year
- Unless the year is also divisible by 400, in which case it is a leap year

Your isLeapYear method should take a single argument of type int representing the year being tested, and should return its answer as a boolean, with true denoting a leap year.

Exercise D.2.13. Write a class IntervalIncludeChecker with static method containsPoint which

- Takes three double values x1, x2, and z
  - The first two parameters x1 and x2 correspond to the start- and endpoints of one interval. We can assume that x1 will be less than or equal to x2.
  - The last parameter z is interpreted as a point on the same axis where the interval [x1, x2] lies.
- Returns an integer value to describe the position of z relative to the interval [x1, x2]:
  - The result should be negative if z is less than x1, that is, if the point lies to the left of the interval.
  - The result should be positive if z is greater than x2, that is, if the point lies to the right of the interval.
  - The result should be zero if z is both greater than or equal to x1 and less than or equal to x2, that is, if the point in the interval.

In both this and the next exercise, assume that the interval is closed, that is, that the endpoints are included in the interval. So for example

- IntervalIncludeChecker.containsPoint(0.0, 2.0, 9.0) should return some positive number.
- IntervalIncludeChecker.containsPoint(0.0, 2.0, 1.0) and IntervalIncludeChecker.containsPoint(0.0, 2.0, 2.0) should both return zero.
- IntervalIncludeChecker.containsPoint(0.0, 1.0, -1.0) should return some negative number.

Exercise D.2.14. Write a class IntervalOverlapChecker with static method intervalsOverlap which

- Takes four double values x1, x2, y1, and y2
  - The first two parameters x1 and x2 correspond to the start- and endpoints of one interval. We can assume that x1 will be less than or equal to x2.
  - The latter two parameters y1 and y2 correspond to the start- and endpoints of a second interval. Again, we can assume that y1 will be less than or equal to y2.
- Returns true when the first and second intervals overlap.

As in the previous exercise, take both intervals to be closed, that is, that the endpoints are included in the intervals. So for example
• `IntervalOverlapChecker.intervalsOverlap(0.0, 2.0, 9.0, 11.0)` and  
  `IntervalOverlapChecker.intervalsOverlap(9.0, 11.0, 0.0, 2.0)` should both return `false`.

• `IntervalOverlapChecker.intervalsOverlap(0.0, 2.0, 1.0, 3.0)` and  
  `IntervalOverlapChecker.intervalsOverlap(1.0, 3.0, 0.0, 2.0)` should both return `true`.

• `IntervalOverlapChecker.intervalsOverlap(0.0, 1.0, 1.0, 2.0)` should return `true`.

□

**Exercise D.2.15.** In these exercises, do not use `Math` methods to calculate the square; just use multiplication.

1. Write a class `SimpleLoop` whose `main` method uses a loop to print the squares of the integers from 0 to 10.

2. Write a class `SimpleLoop2` whose `main` method uses a loop to print the squares of the integers from 10 to 30.

3. Write a class `SimpleLoop3` whose `main` method uses a loop to print the squares of the even integers from 10 to 30.

□

**Exercise D.2.16.** Write a class `SentenceFixer` with a static method `printCapitalized` which

• Accepts a `String` parameter assumed to be a sentence,

• Returns nothing, and

• Prints that sentence making sure the first character is capitalized, and that subsequent characters are lowercase.

The standard methods `toUpperCase` and `toLowerCase` in class `java.lang.Character` will be helpful in converting characters to the correct case. As the usual first step, write a `main` method with examples and expected results. Step through your method by hand for the argument string `HELLO!` to be sure you understand you it works.

□

**Exercise D.2.17.** The factorial function $n!$ is defined informally as $n! = n \cdot (n - 1) \cdot \ldots \cdot 2 \cdot 1$, and is defined formally by two rules:

• If $n = 0$, then $n! = 1$

• If $n > 0$, then $n! = n \cdot (n - 1)!$

Write a class `FactorialFinder` with a static method `factorial` which

• Accepts a single `int` parameter

• Returns a `long` result representing the factorial of the argument.

Since factorial is not defined on negative numbers, it does not matter what your method does for such input. As the usual first step, write a `main` method with examples and expected results.

□
Exercise D.2.18. The choose function from probability is defined as
\[
\binom{n}{m} = \frac{n!}{m!(n-m)!}.
\]
Given the factorial method above, it is certainly possible to extend the FactorialFinder of Exercise [D.2.17] to include a method to implement choose directly:

```java
public static long nChooseM(final int n, final int m) {
    return factorial(n)/factorial(m)/factorial(n-m);
}
```

But this implementation is inefficient, and may cause overflow even when the final result actually can be represented as a long. Write a more efficient version of `nChooseM` which only performs the multiplications and divisions which are absolutely necessary. As usual, extend the `main` method with examples and expected results as a first step.

Exercise D.2.19. The Fibonacci numbers are a sequence of integers indexed from 0 up, defined by:

- Fibonacci number 0 is 0.
- Fibonacci number 1 is 1.
- For any \(n > 1\), Fibonacci number \(n\) is the sum of the two previous Fibonacci numbers (indexed \(n - 1\) and \(n - 2\)).

Write a class `FibonacciFinder` with a static method `fibonacci` which

- Accepts a single `int` parameter \(n\)
- Returns a `long` result representing Fibonacci number \(n\).

Since the series is not defined on negative numbers, it does not matter what your method does for such input. As the usual first step, write a `main` method with examples and expected results.

Exercise D.2.20. Write a class `VowelCounter` with static method `getVowelCount` whose one argument is a `String` and which returns the number of characters in the string which are vowels (a, e, i, o and u).

Exercise D.2.21. The sequence \(A_1\) is defined as follows:

\[
A_1(n) = \begin{cases} 
  n^2 & \text{if } n \text{ is even} \\
  n + 1 & \text{if } n \text{ is odd}
\end{cases}
\]

Write a class `AltSequence` with a static method `getSumTo` which takes an integer \(n\), and returns the sum of the elements 0 through \(n\) of sequence \(A_1\). Use only a single loop, and test inside the loop whether a particular index is even or odd.

Exercise D.2.22. Write a class `ChangeMaker` with a static method `getChange` which

- Takes a single `int` argument representing a number of cents
- Returns a string describing the way to represent that amount in the fewest number of common US coins. The string should be of the form "XX quarters, YY dimes, ZZ nickels, WW pennies". When (for example), YY is one, the string should use the singular dime instead of dimes.
Exercise D.2.23.  What do these loops do? Try to work out what it prints without running it before checking your prediction with Java.

1. for (int i=1; i<=6; i=i+1) {
   for (int j=1; j<=i; j=j+1) {
      System.out.print(i);
   }
   System.out.println();
}

2. for (int i=1; i<=6; i=i+1) {
   for (int j=1; j<=(6-i); j=j+1) {
      System.out.print("-");
   }
   for (int j=1; j<=i; j=j+1) {
      System.out.print(i);
   }
   System.out.println();
}

What if we swap the two inner loops?

Exercise D.2.24.  Write a program to print this triangle:

```
0
01
012
0123
01234
012345
01234
0123
012
01
0
```

Find a solution which uses only a single outer and a single inner loop.

Exercise D.2.25.  Write a class FramedSquare with static method printFramed which

- Takes two arguments frameSize and innerSize
- Draws a square made of asterisks and periods, where
  - The asterisks form a frame on the outer edge of the square with thickness frameSize, and
  - The inside of the frame is filled in with a innerSize-by-innerSize square of periods.

So printFramed(2, 7) would print

```
**********
**********
.........**
.........**
.........**
.........**
**********
```
Exercise D.2.26. Write a class `FrameDrawing` with a static method `drawFramed` which

- Takes two integer arguments
  - The first describing the thickness of a border
  - The second giving the length $n$ of the side of a square
- Draws an $n$-by-$n$ square, with a border of asterisks of the given thickness, and an interior of periods.

So for inputs 2 and 11, your method should print

```
**********
**********
**********
**********
**********
**********
**********
**********
```

D.3 Arrays

Arrays

An array is a primitive data structure for storing multiple objects

- All elements of the array must have the same type
- The length of the array is fixed at its creation, and never changes
- Each position in the array stores a single element
- Each element is referenced by its index in the array

Basic syntax

- Declare an array (does not allocate memory):

  ```java
  final dataType[] arrayName;
  ```
– Alternative syntax

    final dataType arrayName[];

• Allocate memory for a previously declared array:

    arrayName = new dataType[numberOfElements];

Size cannot be negative

• Store and retrieve values in array:

    arrayName[index] = expression; // Store value at index
    arrayName[index]; // Retrieve value from index

• Access the length of an array:

    arrayName.length

Not the same as the method call for String—str.length()

• One-liners for declaration, allocation, and initialization:

    final dataType[] arrayName = new dataType[numberOfElements];
    final dataType[] arrayName = { val1, val2, ..., valN }; // Initialize

Basic examples

• Declare an array (does not allocate memory):

    double[] numbers;    /* Alternate: */ double numbers[];

• Allocate memory for a previously declared array:

    numbers = new double[10];

• Store and retrieve values in array:

    numbers[3] = 7.5;
    System.out.print(numbers[3]);

• Access the length of an array:

    numbers.length

• One-liners for declaration, allocation, and initialization:

    double[] numbers = new double[10];
    double[] numbers = { 1.5, 4.5, 7.5, ..., 15.2 };
Arrays of primitive types
When using arrays, we need to ensure:

• Array variable is declared
• Memory is allocated for the array (using new)
• Contents of the array have been initialized

With primitive type:

```
final int[] intArray = new int[5];
for(int i=0; i<intArray.length; ++i) {
    System.out.print(intArray[i] + ", ");
}
```

• Output:
  
  0, 0, 0, 0, 0,

This works even though we skipped Step 3 – Java takes care of the initialization for us.

Arrays of objects
We can also have arrays of complex type:

```
final Car[] carArray = new Car[5];
for(int i=0; i<carArray.length; ++i) {
    System.out.println(carArray[i]);
}
```

• Output is
  
  null
  null
  null
  null
  null

Why does this fail? Need to initialize array contents!
• Java doesn’t know how to initialize the objects

Example: all civics

```
final Car[] carArray = new Car[5];
// We need to initialize the objects in a sensible way
for(int i=0; i<carArray.length; ++i) {
    carArray[i] = new Car("Honda Civic", 1000 * i);
}
for(int i=0; i<carArray.length; ++i) {
    System.out.println(carArray[i]);
}
```
Arrays of objects

```java
final Person[] simpsons = new Person[3];
simpsons[0] = new Person("Homer", "D’oh!");
simpsons[1] = new Person("Flanders", "Okily Dokily!");
```

In an array of complex type (i.e., class), each element in the array stores a reference to an object of that class

- Does not store the object itself (just like a variable of complex type)
- We need to instantiate an object for each element of the array

**Exercise D.3.1.** Trace through the execution of this class (without running it first). What does it print?

```java
public class UseAnArray {
    public static void main(String[] argv) {
        int[] numbers = {10, 20, 30, 40};  // 1
        for(int i=0; i<numbers.length; i=i+1) {  // 2
            final int number = numbers[i];  // 3
            System.out.println(number);  // 4
        }
    }
}
```

**Exercise D.3.2.** What is the largest Java array you can define? Is there a difference between the largest array that the Java compiler will allow you define, and the largest array that a running program on the system you happen to be using right now can allocate to a running program?

**Exercise D.3.3.** What errors do you get when you:

- Try to read from an array slot which is beyond the upper limit of an array?
- Use round parentheses instead of square brackets when you declare an array?
- Use round parentheses instead of square brackets when you access an array element?

Which of these errors are compile-time errors, and which are run-time errors?

**Exercise D.3.4.** Write a class `UseAStringArray` whose main method

- Declares a String array containing two values, "Hello" and "Goodbye", and then
- Loops through the array to print each of the values.

**Exercise D.3.5.** Update your class `MonthNamer` from Exercise D.2.3 to use an array within `getMonthName`. 
Exercise D.3.6. Write a class ColumnMaker with a static method printInColumn which

- Takes an array of integers as its single argument, and
- Prints the numbers right-aligned in a single column.

Use your getCharacterLength method from Exercise D.1.8.

□

Exercise D.3.7. Write a class StatsFinder with a static method printSummaryStats which takes an array of double values as its single argument, and calculates and prints messages detailing

- How many numbers there are.
- Their mean: the sum of the values divided by how many values there are.
- Their standard deviation: \( \sigma = \sqrt{\frac{\sum (\bar{x} - x_i)^2}{N - 1}} \), where \( \bar{x} \) is the mean and \( N \) is the number of values.

□

Exercise D.3.8. Extend method printSummaryStats of Exercise [D.3.7](#) to also print the median: the middlemost value contained in the array.

□

Exercise D.3.9. Modify StatsFinder from Exercise [D.3.7](#) (or [D.3.8](#) or [D.3.18](#)) to work with integers, and to additionally calculate:

- The maximum and minimum values of the array.
- The number of different values in the array, and how many times each one appears in the array.
- Their mode, the value which appears more often than any other.

□

Exercise D.3.10. Write a class ArrayIntFinder with a static method getIndexOf which

- Takes two arguments
  1. An array numbers of integers, and
  2. A single integer target
- Searches numbers for the index where target is found
- Returns either that index, or -1 if target was not found in numbers

When testing your code, be sure to try both cases where target is found in numbers, and cases where target is not found in numbers.

□

Exercise D.3.11. Write a class SortedArrayIntFinder with a static method getIndexOf which whose arguments and result are just as in class ArrayIntFinder of Exercise [D.3.10](#) but where your method is allowed to assume that the array is sorted in order. Take advantage of the assumption of an ordered list by starting your search in the middle of the list, so that after every comparison your method can exclude half of the unsearched elements based on the target being greater or less than the middle element.

(For a hint, see p. [229](#))
Exercise D.3.12. The dot product is a common mathematical operation on pairs of numeric vectors (arrays) of the same size. Given two vectors \((x_i)_{n1}\) and \((y_i)_{n1}\), their dot product \(\vec{x} \cdot \vec{y} = \sum_{i}^{n} x_i y_i\). Write a class DotProduct with static method getDotProduct which

- Takes two arguments of type double[], and
- Returns the double dot product of the two arrays.

It does not matter what your program does if the arrays are not the same size. As the usual first step, write a main method with examples and expected results.

Exercise D.3.13. Trace through the execution of this class (without running it first). What does it print?

```java
public class ChangeAnArray {
    public static void main(String[] argv) {
        int[] numbers = { 10, 20, 30, 40);

        for(int i=0; i<numbers.length; i=i+1) {
            numbers[i] = numbers[i] + 2;
            System.out.println(numbers[i]);
        }
    }
}
```

Exercise D.3.14. What errors do you get when you:

- Try to write to an array slot which is beyond the upper limit of an array?
- Assign a String to an array which you have declared to hold int values?

Which of these errors are compile-time errors, and which are run-time errors?

Exercise D.3.15. What do these programs do? Trace through the programs by hand before running them in Eclipse to confirm your hypotheses.

```java
public static void main(String[] args) {
    int y=30;
    f(y);
    System.out.println(y);
}

public static void f(int x) {
    System.out.println(x);
    x=10;
    System.out.println(x);
}
```

```java
public static void main(String[] args) {
    int x=30;
    f(x);
    System.out.println(x);
}
```

```java
public static void f(int x) {
```
Exercise D.3.16. What do these programs do? Trace through each one by hand before running them in Eclipse to confirm your hypotheses.

- public static void main(String[] args) {
   int[] y = new int[] { 10, 20, 30 };  
f(y);  
for(int i=0; i<y.length; i=i+1) {
   System.out.println(y[i]);
}
}

public static void f(int[] x) {
   if (x.length > 1) {
      x[1] = -1;
   }
   return;
}

- public static void main(String[] args) {
   int[] y = new int[] { 10, 20, 30 };  
f(y);  
System.out.println("==========");
for(int i=0; i<y.length; i=i+1) {
   System.out.println(y[i]);
}
}

public static void f(int[] x) {
   x = new int[] { 1, 2, 3, 4, 5 };  
x[1] = -1;
for(int i=0; i<x.length; i=i+1) {
   System.out.println(x[i]);
}
   return;
}

- public static void main(String[] args) {
   int[] y = new int[] { 10, 20, 30 };  
f(y);  
System.out.println("==========");
for(int i=0; i<y.length; i=i+1) {
   System.out.println(y[i]);
}
}

public static void f(int[] x) {
   for(int i=0; i<x.length; i=i+1) {
      x[i] = i*100;
   }
}
for(int i=0; i<x.length; i=i+1) {
    System.out.println(x[i]);
}
x = new int[] { 1, 2, 3, 4, 5 };
if (x.length > 1) {
    x[1] = -1;
}
System.out.println("==========");
for(int i=0; i<x.length; i=i+1) {
    System.out.println(x[i]);
}
return;

Exercise D.3.17. Write a class \texttt{ScaleBy} with method \texttt{scaleByFactor} which

- Takes two arguments:
  1. An array \texttt{numbers} of \texttt{double}
  2. Another \texttt{double} value called \texttt{factor}
- Multiplies every element of \texttt{numbers} by \texttt{factor}
- Has no explicit return value

Include a main routine with several tests of \texttt{scaleByFactor}

Exercise D.3.18. Add another static method \texttt{getSummaryStats} to class \texttt{StatsFinder} from Exercise [D.3.7]. Your \texttt{getSummaryStats} should, instead of just printing the various statistics, returns a new \texttt{double[]} array where element 0 is the mean, element 1 is the standard deviation, and so on. As the usual first step, create several tests as the main method of \texttt{StatsFinder}. Be sure to examine the length of the array as well as each element. Rewrite \texttt{printSummaryStats} to remove duplicated code, so that it just calls \texttt{getSummaryStats} and prints its results in a comprehensible manner.

Exercise D.3.19. Write a class \texttt{CapitalizeAllChars} with method \texttt{upcaseAll} which

- Takes two arguments:
  1. An array \texttt{characters} of \texttt{char}
- Capitalizes every element of \texttt{characters} (use the library method \texttt{Character.toUpperCase})
- Has no explicit return value

Include a main routine with several tests of \texttt{upcaseAll}

Exercise D.3.20. Write a class \texttt{TwoGrouper} with method \texttt{sortIntoGroups} which

- Takes a single argument, an array of characters, which your method should assume contains only 'R' and 'W', and
- Reorganizes the array so that all of the 'R' s come before all of the 'W' s.

\textit{Important restriction}: your program may "visit" each element of the list only once. So for example, a program which simply goes through the array once to count the 'R' s, and which then makes a second pass to assign 'R' s and 'W' s, would not satisfy this restriction!

(For a hint, see p. 229)
Exercise D.3.21. (Continues from Exercise D.3.20) Write a class ThreeGrouper with method sortIntoGroups which

- Takes a single argument, an array of characters, which your method should assume contains only ‘R’, ‘W’ and ‘B’, and
- Reorganizes the array so that all of the ‘R’ s come before all of the ‘W’ s, all of the ‘W’ s come before all of the ‘B’ s.

Follow the same restriction as for Exercise D.3.20: visit each node only once.

This problem was invented by Edsgar W. Dijkstra, who discovered many important computer science algorithms, and who originated in the Netherlands. The Dutch flag consists of three stripes, one red, one white, and one blue, and Dijkstra named this problem *The Dutch National Flag problem*.

Exercise D.3.22. Write a class ArrayReverser with a static method reverse which

- Takes a single argument, an array of *String* values, and
- Returns a new array which contains the same elements as the argument, but in the reverse order.

The array used as an argument should not be altered in any way by the reverse method.

Follow the usual procedure for writing stubs, and then developing example calls and a design, before starting the Java implementation.

Exercise D.3.23. Write a class ArraySplitter with a static method getUpperHalf which

- Takes a single argument, an array of *int* values
- Return an array which
  * Is half the length of the argument array
  * Contains the same values as the elements of the argument from the midway point to the highest index.

As usual, begin with a main method containing tests which verify the behavior of getUpperHalf.

Exercise D.3.24. Write a class TakeEveryThird with a static method everyThird which

- Takes a single argument, an array of *int* values, and
- Returns an array which contains only every third element of the original, beginning with the first one.

So a call to

```java
final int[]
  result = everyThird(new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 });
```

for(int i=0; i<result.length; i++) {
  System.out.println(result[i]);
}
```

should print:

```
2
8
14
20
```
**Exercise D.3.25.**  Write a class `TakeEveryNth` with a static method `everyNth` which

- Takes two arguments,
  1. An integer
  2. An array of `int` values
- Returns an array which contains only every \( n \)th element of the array argument, where \( n \) is the integer argument.

So a call to

```java
final int[] result = everyNth(3, new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 });
for(int i=0; i<result.length; i++) {
    System.out.println(result[i]);
}
```

would print the same output as the example for Exercise D.3.24 and a call to

```java
final int[] result = everyNth(5, new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36 });
for(int i=0; i<result.length; i++) {
    System.out.println(result[i]);
}
```

should print:

```
2
12
22
32
```

**Exercise D.3.26.**  Write a class `ArrayInterleaver` with a static method `interleaveArrays` which

- Takes two arguments, each an array of `String` values
- Returns a new array
  * Whose length is the sum of the two argument arrays
  * And which contains the elements of the arrays, drawing in alternation from one array and then the other. If one argument array is longer than the other, then its additional elements should appear together at the end of the result.

So for arguments containing

```
"A", "B", "C"
```

and

```
"V", "W", "X", "Y", "Z"
```
the result should contain


As usual, begin with a main method containing tests which verify the behavior of `interleaveArrays`.

**Exercise D.3.27.** The main method of your class `ArrayReverser` for Exercise D.3.22 probably contains example calls of `reverse` which look like this:

```
System.out.println("For { "A", "B", "C", "D", "E" } ");
System.out.println("reverse should return { "E", "D", "C", "B", "A" } ");
System.out.println("actually returns { ");
final String[] result = reverse(new String[] { "A", "B", "C", "D", "E" });
for(int i=0; i<result.length; i=i+1) {
    if (i>0) {
       System.out.print(",");
    }
    System.out.print(" " + result[i]);
}
System.out.println(" }");
```

When we have multiple similar tests, we will end up with a considerable amount of duplicated code. But this is exactly why we have methods! The code for different tests of `reverse` will differ only by the actual argument and expected result — the rest will be the same from test to test. So we can write a `testReverse` method (note that this is not the same as the `reverse` method!) to consolidate all of the similar code into one method as shown in Figure 1 on page 218. With this method in place, we can replace the test code with just a call to `testReverse`

```
    testReverse(
        new String[] { "A", "B", "C", "D", "E" },
        new String[] { "E", "D", "C", "B", "A" });
```

Adopt this idea to other problems from this section:

1. For Exercise D.3.23 write a method `testGetUpperHalf` in class `ArraySplitter` for one test call of method `getUpperHalf`. Either replace your tests in the main method with similar calls to `testGetUpperHalf`, or add new tests for it.

2. For Exercise D.3.24 write a method `testEveryThird` in class `TakeEveryThird` for one test call of method `everyThird`. Either replace your tests in the main method with similar calls to `everyThird`, or add new tests for it.

3. For Exercise D.3.25 write a method `testEveryNth` in class `TakeEveryNth` for one test call of method `everyNth`. Since `everyNth` takes two arguments, `testEveryNth` will need to take three arguments. Either replace your tests in the main method with similar calls to `everyNth`, or add new tests for it.

4. For Exercise D.3.26 write a method `testInterleaveArrays` in class `ArrayInterleaver` for one test call of method `interleaveArrays`. Either replace your tests in the main method with similar calls to `interleaveArrays`, or add new tests for it.
Exercise D.3.28. We can further improve the testing of reverse from Exercise [D.3.27](#) by adding a method to print the contents of one array, and replacing the repeated loops in the test method with a call to this second new helper method.

1. In the ArrayReverser class as extended in Exercise [D.3.27](#), add a method `printContents`:

   ```java
   public static void printContents(String[] arr) {
     System.out.println("{";
     for(int i=0; i<arr.length; i=i+1) {
       if (i>0) {
         System.out.print(",");
       }
       System.out.print(" "+arr[i]);
     }
     System.out.println(" }") ;
   }
   ```

   Simplify `testReverse` to use this new method instead of repeating the iteration multiple times. Make sure the output of `main` is unchanged by this simplification.

2. Apply a similar simplification in class ArraySplitter from Exercise [D.3.27](#)(a).
3. Apply a similar simplification in class TakeEveryThird from Exercise [D.3.27](#)(b).
4. Apply a similar simplification in class TakeEveryNth from Exercise [D.3.27](#)(c).
5. Apply a similar simplification in class ArrayInterleaver from Exercise [D.3.27](#)(d).

Exercise D.3.29. Write a class `WordsTaker` with a static method `getWords` which

- Takes two arguments,
  1. A `String`, expected to consist of several space-separated words
  2. An `int`, representing how many of these words are of interest
- Returns an array of strings
  * The length of the array should be the same as the integer argument
  * The first (index 0) element of the result array should be the first space-separated word of the string argument, and so on

Assume for this exercise that there will always be enough words in the string for the integer argument. As the usual first step, create several tests of `getWords` for the `main` method of `WordsTaker`. Be sure to examine the length of the array as well as each element.

Exercise D.3.30. Write a class `BiggestElements` with static method `getBiggest` which

- Takes two arguments
  1. An array of integers `numbers`
  2. An additional integer `howMany`
- If `howMany` is less than the length of `numbers`, then `getBiggest` should return a new array of length `howMany` containing the biggest values `numbers`.
  If `howMany` is greater than or equal to the length of `numbers`, then it should return `numbers` as-is.

As usual, begin with a `main` method containing tests which verify the behavior of `getBiggest`. □
Exercise D.3.31. *Selection sort* implements the following idea for sorting an array:

- First, go through every element of the list, and find the index of the smallest value. If that value is not at slot 0, swap it with the value at slot 0.
- Next, go through every element of the list except the first, and find the index of the smallest value among them. If that value is not at slot 1, swap it with the value at slot 1.
- And so on, for each slot in the list.

Write a class `SelectionSorter` with a static method `sort` which

- Takes an array of integers as its input,
- Performs a selection sort on the array, and
- Returns nothing.

Follow our usual process of stubbing methods, creating several tests of `sort` for the `main` method of `SelectionSorter`, and adapting the above idea into a design, before implementing `sort` in Java code.

Exercise D.3.32. Write a class `FilteredSumOfSquares` with a single static method `getFiltered` which

- Takes two parameters,
  1. An array of `int` values called `vals`, and
  2. An `int` value called `threshold`
- Returns an `int` value, the sum of the squares of the values in `vals` which are larger than `threshold`.

So for example, a call to

```
getFiltered(new int[] { 2, 5, 10, 1 }, 4)
```

should return `125`, since 5 and 10 are larger than 4, but 2 and 1 are not.

D.4 Classes and objects

Real-life objects

This object is commonplace, and yet complicated

- It possesses some *state*
  - Including its current location, gear, current speed
- It has some *behaviors*
  - Like moving, accelerating, braking
- It *interacts with* other objects
  - Like the road, other cars, trees, people
- It is *made of* other objects
  - Like the engine, seats, tires, radio

Most of us can use it as a *black box*
public static void testReverse(String[] argument, String[] expected) {
    System.out.println("For ");
    for(int i=0; i<argument.length; i=i+1) {
        if (i>0) {
            System.out.print(",");
        }
        System.out.print(" " + argument[i]);
    }
    System.out.println(" ");
    System.out.println("reverse should return ");
    for(int i=0; i<expected.length; i=i+1) {
        if (i>0) {
            System.out.print(",");
        }
        System.out.print(" " + expected[i]);
    }
    System.out.println(" ");
    System.out.println("actually returns ");
    final String[] result = reverse(argument);
    for(int i=0; i<result.length; i=i+1) {
        if (i>0) {
            System.out.print(",");
        }
        System.out.print(" " + result[i]);
    }
    System.out.println(" ");
}

Figure 1: Method testReverse for Exercise D.3.27.
• Don’t need to understand how it works
• Just need to know what we can do with it

Running a Java program
A CarLot will create Car objects, but what creates the CarLot itself?
The solution is to use static methods (and variables).
• Content marked static is independent of any object instance
• Usually associated with the class itself

```java
public class CarLot {
p油画 Car car1;
p油画 Car car2;

public static void main(String[] args) {
    CarLot myCarLot = new CarLot();
}

public CarLot() {
    car1 = new Car("Honda Civic", 214118);
    ...
}
```

The main method
In Java, the main method has special significance
• Provides a point of entry for starting a program
  – Must be public and static
• Any class can have a main method
• Must have proper signature (including String array param)
• In OO paradigm, main typically creates a top-level object and invokes a method which then takes over

```java
public class CarLot {
    ...
    public static void main(String[] args) {
        CarLot myCarLot = new CarLot();
        myCarLot.manage();
    }
    ...
    public void manage() {
        // Most of program functionality goes here
    }
}
```
Exercise D.4.1. Create a class `Student` to store the following information about a student:

- Their name
- Their ID
- Their current classes
- The number of credits earned

Include a constructor which receives these values, and accessor methods for each.

- Store their name in two fields `firstNames` and `lastName`
- Store their ID as an `int`
- Store their current classes as an array of strings
- Store the number of credits earned as an `int`

Exercise D.4.2. Create a class `Vehicle` with the following private field:

- A single `String` field `makeModel` for the make/model
- An `int` field `fuelCap` for the capacity of the fuel tank
- A `double` field `mpg` for the miles-per-gallon efficiency

Define a constructor which receives and assigns the field values in the above order, and accessor methods for each.

Exercise D.4.3. Create a class `SSN` to store a Social Security number. Class `SSN` should have:

- Three private and `final` fields `group1`, `group2` and `group3`, each of type `int`, corresponding to the three parts of a Social Security number
- Accessors for these three fields
- A method `ssnToString` which takes no arguments, and formats the SSN in the usual way with a hyphen between fields

Include two constructors for `SSN`

- One constructor takes a single `int` argument representing the entire SSN.
- The other constructor takes three `int` arguments representing the three segments.

Each constructor should set the fields `group1`, `group2` and `group3` in the way appropriate for its arguments.

Exercise D.4.4. Change class `SSN` to have a *single* private field `number`, an `int` representing the entire nine-digit SSN. The methods (including `getGroup1` and so forth) should behave just the same way.
Exercise D.4.5.  Extend class Student of Exercise [D.4.1] with a method hasUnitsToGraduate which

• Takes no arguments, and

• Returns true if the student has earned the 120 units needed to graduate.

Exercise D.4.6.  Add a method getRange to class Vehicle of Exercise [D.4.2] which calculates from the the
fuelCap and mpg fields the maximum distance which the vehicle could drive on a single tank. As the usual first
step, add tests of getRange to a main method.

Exercise D.4.7.  Write a class QuizScores to hold information about a student’s quiz scores in a class (obvi-
ously not this class, since we do not have quizzes). Your class should have the following constructor and methods:

• The constructor should take an array of integers, holding the quiz scores.

• Methods getHighest and getLowest should return the highest and lowest scores received. They take
no arguments, and return an int.

• Method getAverage should return a double value, the average score received.

Exercise D.4.8.  Extend class QuizScores from Exercise [D.4.7] with two additional methods,

• Method countGreaterThan takes an integer and returns the number of quiz scores which are greater
than the argument.

• Method getAverageDroppingLowest should return a double value, the average score excluding the
lowest.

As the usual first step, add tests of these methods to a main method.

Exercise D.4.9.  Write a class CharChecker whose constructor takes a single string (which we will refer to as
base), and with the following methods:

• An accessor getBase for base

• A method countMatches which take a single string str, and which returns the number of characters of
str which are also in base.

For example, the statements
CharChecker cc = new CharChecker("abc");
System.out.println(cc.countMatches("abracadabra"));

should print 8.

As the usual first step, add tests of these methods to a main method. Do not use any methods of String
except charAt and length to solve this problem.

StateFinder implementations before starting these exercises, and expect to deploy some of the same algo-
rithms here.
Exercise D.4.10. Write a class `DataSet` to perform statistical analysis of a set of numbers.

- It should have a constructor which takes a single argument, an array of `double` values.
- Method `mean` should take no arguments, and should return the mean of the values from the array.
- Its field for storing data should be private.

As the usual first step, add tests of its method to a `main` method.

Exercise D.4.11. Consider your class `DataSet` from Exercise D.4.10. What happens if we run

```java
int[] numbers = new int[] { 5.0, 6.0, 7.0 };
DataSet ds = new DataSet(numbers);
numbers[1] = 12.0;
System.out.println(ds.mean());
```

With the usual setup for fields

```java
private double[] numbers;
public DataSet(double[] numbers) {
    this.numbers = numbers;
}
```

The `mean` would return 8.0, not 6.0 — the array itself is shared — we have allocated only one of them, and passed around references to it. Update the `DataSet` constructor to make sure that the storage we use for the numbers in a `DataSet` instance is separated from the array passed to the constructor. (See p. 229 for a hint if you need.)

Exercise D.4.12. Extend class `DataSet` of the previous exercises to give the data sets a name.

- Add an additional constructor that takes both the `String` name and a `double` array.
- Provide accessor and mutator methods `getName` and `setName`.
- But keep the original constructor as well! For a `DataSet` instance created without a name, the `getName` accessor should return the string "(unnamed)".

Exercise D.4.13. Add a method `stdDev` to `DataSet` from Exercise D.4.17 which takes no arguments and returns the standard deviation of the data. (See Exercise D.3.7 for a formula for standard deviation.) Since we use the mean for calculating the standard deviation, `stdDev` will need to call `mean`.

Exercise D.4.14. Extend class `Student` of Exercise D.4.1 with the assumption that a student ID will never change.

Exercise D.4.15. Extend class `Student` of Exercise D.4.14 with methods to update a student record:

- Update the name
- Report passed classes
- Enroll in new classes
Exercise D.4.16. Revise class `Vehicle` of Exercise D.4.6 with an additional private fields:

- An `int` field `mileage` for the number of miles recorded by the vehicle

Modify the constructor to initialize this field as well, and add both an accessor and a mutator method to set the field. As the usual first step, add tests of using the setter to a main method.

Exercise D.4.17. Extend the class `DataSet` of Exercise D.4.10 with two methods:

- `addData`, which takes an array `additionalData` of double values. These values should be added to the values already in the `DataSet`, and should cause `mean` to return an updated, new value based on all of the data.
- `clearData`, which resets the data storage to contain no numbers.

Exercise D.4.18. Consider your class `DataSet` from Exercise D.4.17 (without the `stdDev` method of Exercise D.4.13). We have made `mean` a method, but when the data do not change, the mean will not change either. Update the class and methods so that we recompute the mean value only when we need to, and just save the value when the data is unchanged. (See p. 229 for a hint if you need.)

Exercise D.4.19. Redo Exercise D.4.18 but starting from Exercise D.4.13 instead of Exercise D.4.17 (or alternatively, add a `stdDev` method to your result of Exercise D.4.18). Cache the two values separately, updating each one only when demanded.

Exercise D.4.20. Another way to add caching of derived values is to use a single boolean flag to indicate new data, but still multiple double fields for the cached values, plus a private helper method `updateCached` which recomputes all of the cached values. Extend `DataSet` of Exercise D.4.13 to implement this idea, making sure that `mean` and `stdDev` call it only when there is new data.

Exercise D.4.21. In `DataSet`, we considered data where the individual points have no particular identity — they were just numbers. Sometimes, individual data points have an identity beyond just their value, and each comes with its own label. For examples, the times of racers may be labeled with the racer’s name, or rainfall totals may be labelled with the location of measurement.

Write a class `TopThree` to keep track of the three highest labelled data values. There should be six simple accessor methods for the stored data:

- Methods `getValue1` and `getLabel1` return the highest data value, and its label
- Methods `getValue2` and `getLabel2` return the second-highest data value, and its label
- Methods `getValue3` and `getLabel3` return the third-highest data value, and its label

There should be one mutator method `newResult` which

- Takes two arguments, a `double` value and a `String` label
- Returns no value
- Might update some of the top three values, if the value is higher than the ones reviewed previously

The constructor for `TopThree` should take the initial top three values and labels

```java
public TopThree(double bestValue, String bestLabel,
                double secondValue, String secondLabel,
                double thirdValue, String thirdLabel)
```

Modify class `TopThree` of Exercise D.4.21 so that the constructor does not assume that the three value/label pairs are already in order, but instead works out for itself which label has the highest value.
Exercise D.4.22. Write a class `Complex` to model complex numbers

- The constructor should take two `double` values, for the real and imaginary parts
- Numbers are constants, so the fields should be final
- Provide accessors `getReal` and `getImaginary`
- Provide method `magnitude`, returning a `double`
- Provide method `conjugate`, which needs no arguments, and returns another `Complex`
- Provide methods `add`, `subtract`, `multiply` and `divide`, each taking a single `Complex` argument and returning a `Complex` result

The formulas for these operations are on p. 229.

Exercise D.4.23. In Exercise D.4.17 we added a method `addData` which changed the underlying data, so that `mean` would normally return a different value. For this exercise we will again extend the `DataSet` Exercise D.4.10 to add data, but in a way that does not change the underlying data set:

- Add a method `addData` which takes an array of `double` values, and returns a new `DataSet`. The resulting instance should contain all of the values of the original instance, plus all values from the argument array. The original instance should not be changed in any way.

- Add a method `removeData` which takes an array of `double` values, and which also returns a new `DataSet`. The resulting instance should contain all of the values of the original instance except the values from the argument array. Again, the original instance should not be changed in any way.

As usual, begin by adding tests of the new methods to `main`.

Exercises D.4.24 and D.4.25 refer to class `Rectangle`:

```java
public class Rectangle {
    private final double width, height, leftX, upperY;

    /**
     * Construct the representation of a rectangle on the plane.
     * @param width The width along the x-axis of this rectangle
     * @param height The width along the y-axis of this rectangle
     * @param leftX The x-coordinate of the left edge of the rectangle
     * @param upperY The y-coordinate of the upper edge of the rectangle
     */
    public Rectangle(final double width, final double height,
                     final double leftX, final double upperY) {
        this.width = width;
        this.height = height;
        this.leftX = leftX;
        this.upperY = upperY;
    }

    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public double getLeftX() { return leftX; }
    public double getUpperY() { return upperY; }
}
```
Exercise D.4.24.  Add a method `overlaps` to class `Rectangle` which

- Takes one argument, which is also a `Rectangle`, and
- Return `true` when the two rectangles overlap in the plane with an area greater than 0, or `false` otherwise.

As usual, write a `main` method for `Rectangle` which verifies that `overlaps` is implemented correctly.

Exercise D.4.25.  Add a static method `centeredAt` to class `Rectangle` which

- Takes four `double` arguments,
  1. A width
  2. A height
  3. An x-coordinate
  4. A y-coordinate, and
- Returns a `Rectangle` whose `center` is at the given point.

Extend the `main` method for `Rectangle` to verify that `centeredAt` is implemented correctly.

Exercise D.4.26.  In algebra we study two ways of representing points on a plane: for `Cartesian coordinates` we give two values representing distance along the x- and y-axes, and for `polar coordinates` we give a distance from the origin or `magnitude`, and an angle offset counterclockwise from the x-axis. The angle value is usually given in radians, where $2\pi$ is the measure of a single full rotation around the circle.

This interface describes operations we might want to associate with a point:

```java
public interface Point {
    public double getX();
    public double getY();
    public double getMagnitude();
    public double getAngle();
    public Point addVectors(Point that);}
```

Create two classes `CartesianPoint` and `PolarPoint` which implement this interface, and which represent points (both in their constructor and in the fields of each object) in these two ways. Where we understand a point be a vector from the origin to a coordinate, the `addVectors` methods returns the point we find when we follow both vectors, first one and then the other (for a hint, see p. [229]). Make sure that the result of `CartesianPoint`’s version of `addVectors` is again a `CartesianPoint`; and the result of `PolarPoint`’s version, a `PolarPoint`.

Exercise D.4.27.  There is another common way to represent complex numbers. We can imagine the complex numbers as points on a plane, with the real component for the x-axis, and the imaginary component as the y-axis. But we can also represent points on the plane by two other numbers: their distance from the origin, and the measure of the angle formed by a line from the point to the origin, and the positive part of the x-axis.

So for example, the point $(1,1)$ is at a distance of $\sqrt{2}$ from the origin. It forms a $45^\circ$-angle with the positive part of the x-axis, but we use radian measure instead of degrees, so we would measure the angle as $\frac{\pi}{4}$. In terms of complex numbers, we write: $1 + i = \sqrt{2}e^{\frac{\pi}{4}i}$.

Refactor the class `Complex` of Exercise D.4.22 to have the name `CartesianComplex`, and instead define `Complex` to be this interface:

```java
public interface Complex {
    public double getReal();
    public double getImaginary();
    public boolean isZero();
}
```
Then create a class PolarCartesian which also implements Complex. Be sure you use accessor methods to access the components of a number passed as an argument to the various operations — this way it makes sense to add two numbers which happen to have different representations.

As the usual first step, write tests making sure that both CartesianComplex and PolarComplex are correct.

Exercise D.4.28. Write a class Square which extends class Rectangle of Exercises D.4.24 and D.4.25. Square should have a single constructor which takes three double arguments: the first for the length of a side of the square, and the other for the x- and y-coordinates of the upper-left corner of the square.

Extend the main method for Rectangle to verify that your constructor for Square is implemented correctly.

Exercise D.4.29. In Exercise D.1.8 we thought about the length of a number. We assumed that numbers should be written in base 10 — that is, with digits from 0 through 9, and place values corresponding to powers of 10. So the number we write as 6,098 means $6 	imes 10^3 + 0 	imes 10^2 + 9 	imes 10^1 + 8 	imes 10^0$.

But we can use other bases too: the number we write as 41 in base 10 would be written as 101001 in base 2 ($1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$), and in base 7 as 56 ($5 \times 7^1 + 6 \times 7^0$). We can think of bases larger than 10 as well. Base 12 usually uses digits A for 10 and B for 11, so 41 in base 10 would be 35 ($3 \times 12^1 + 5 \times 12^0$), and 1,575 in base 10 would be ab3 ($10 \times 12^2 + 11 \times 12^1 + 3 \times 12^0$).

Write a class BaseRepresenter for finding the String representation of numbers in a particular base, with:

- A single method numberToString, which
  - Takes a single long argument, and
  - Returns the String representation of the argument in the object’s base; and

- A constructor which takes a single argument of type char[], giving the digits to be used in representing numbers (and implicitly, the place value to be used).

Then, write classes InBase2 and InBase16 which are subclasses of BaseRepresenter, but which only define a constructor, and do not need to override numberToString.

Exercise D.4.30. Update class SSN from Exercise D.4.3 so that, instead of defining method ssnToString it overrides toString to implement that behavior.

Exercise D.4.31. The class Card models playing cards

```java
public class Card {
    private final String suit;
    private final int rank;

    public Card(final String suit, final int rank) {
        this.suit = suit;
        this.rank = rank;
    }
}
```
public String getSuit() { return suit; }
public int getRank() { return rank; }

Add an equals method to this class which overrides Object.equals, and which returns true when the two Card instances have both the same rank and the same suit. Remember to think about the appropriate comparison techniques for each field, and add tests of the correct operation of equals to your main method.

Exercise D.4.32. Add a toString method to class Card which overrides Object.toString, and which

- Returns an output string of the form RANK of SUIT. So for example the statements

  Card c = new Card("hearts", 5);
  System.out.println(c);

  should print 5 of hearts

- Prints rank 1 as ace, rank 11 as jack, rank 12 as queen, and rank 13 as king. So for example the statements

  Card c = new Card("spades", 12);
  System.out.println(c);

  should print queen of spades. Add tests to ensure that toString returns correct results to your main method.

Exercise D.4.33. Add a method equals to class Complex of Exercise D.4.22 which overrides Object.equals, and which equates two instances of Complex exactly when they have both the same real component, and the same complex component. Add tests to your main method to ensure that equals returns correct results.

Exercise D.4.34. Add a toString method to class Complex of Exercise D.4.22 which overrides Object.toString, and which

- Returns the single-character 0 for any instance in which both components are zero
- Returns the string representation of the real component only when the complex component is zero
- Returns the string representation of the imaginary component followed by i when the real component is zero
- Returns a string of the form X+Yi otherwise

Add tests to your main method to ensure that toString returns correct results.

Exercise D.4.35. Add toString methods to classes CartesianComplex and PolarComplex of Exercise D.4.27 which override Object.toString, and which both return a string in the same format as described in Exercise D.4.34. Add tests to your main methods to ensure that toString returns correct results.

Exercise D.4.36. Add equals methods to classes CartesianComplex and PolarComplex of Exercise D.4.27 which override Object.equals, and which equates two instances of classes implementing Complex exactly when they denote the same complex number. Add tests to your main method to ensure that equals returns correct results.
Exercise D.4.37. Write classes which implement a media library:

- Class **Media** includes immutable private fields and public accessors for the title, primary artist, and playing time
- Classes **Video** and **Audio** both extend **Media**
  - Class **Video** includes immutable private fields and public accessors for the director, studio and actors
  - Class **Audio** includes immutable private fields and public accessors for the music company/label
- Classes **DVD**, **FromNetflix**, **CD**, and **Cassette** extend the above classes with further information specific to each format

D.5 GUIs

Most CS120 sections include a unit on Java Swing programming. There are many, many resources for learning/reviewing Swing programming online, and we do not attempt to reproduce them here.

Because not all CS220 students will have covered Swing in their non-UWL equivalents of CS120, we use a number of prepared classes to simplify GUI programming.

The **DrawingPad** class provides a simple interface for drawing lines and shapes in a graphical window. You can download this class from

https://cs.uwlax.edu/~jmaraist/u/220/code/DrawingPad.java

**Exercise D.5.1.** Download the **DrawingPad** class, and set up a new Eclipse project which includes that class as a source file. If you have not yet discovered how to view the Javadoc pages generated from a class, do so. Generate the Javadoc page for **DrawingPad**, and read about the methods which it provides.

**Exercise D.5.2.** Write a class **ItsAPuppy** which uses the **DrawingPad** class to sketch a puppy.

**Exercise D.5.3.** Write a class **GraphSquared** which uses the **DrawingPad** class to draw an x- and y-axis with the plot of the function $y = x^2$. 

229
E  Hints for selected exercises

E.1 Review exercises

Exercise D.1.8  Use the logarithm function for base 10.

Exercise D.3.20  Use two different \texttt{int} variables as two different pointers into the list — one from the start, looking for \texttt{Ws}; the other from the end, looking for \texttt{Rs}. Use a \texttt{while} loop instead of a \texttt{for} loop to decide whether there are more swaps to be made.

Exercise D.3.11  

- Identify \texttt{lo} and \texttt{hi} indices
- Structure your searching loop to continue as long as \texttt{lo} < \texttt{hi}.
- When \texttt{target} is less than what you find at the midpoint between \texttt{lo} and \texttt{hi}, update \texttt{hi} to exclude the indices where you now now that \texttt{target} could not possibly be; and similarly for when \texttt{target} is greater than the middle element.

Exercise D.4.11  Allocate a new array for the \texttt{DataSet} object, and then copy the numbers into it.

Exercise D.4.18  Add a pair of \texttt{private} fields:

- A \texttt{boolean} field \texttt{newData} which set to \texttt{true} whenever there is new data.
- A \texttt{double} field \texttt{mean} which is sometimes set to the mean value.

The method \texttt{mean} should check whether there is new data: if not, then it can used the stored value; if so, it should first update both of these two fields.

Exercise D.4.22  

- Magnitude: $|x + iy| = \sqrt{x^2 + y^2}$
- $(x + iy) + (z + iw) = (x + z) + i(y + w)$
- $(x + iy) - (z + iw) = (x - z) + i(y - w)$
- $(a + bi) \cdot (c + di) = ac + adi + bci - bdii = (ac - bd) + i(ad + bc)$
- Conjugate: $a + ib = a - ib$
- $\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{(c+di)(c-di)}$. Note that the product of a number with its conjugate is a real number.

Exercise D.4.26  Most of the math in this exercise is straightforward trigonometry, but the formula for adding two vectors specified as polar coordinates is more complicated. Given two points $(r_1, \theta_1)$ and $(r_2, \theta_2)$, the vector of their sum is $(r, \theta)$ where:

$$r = \sqrt{r_1^2 + r_2^2 + 2r_1r_2\cos(\theta_2 - \theta_1)}$$
$$\theta = \theta_1 + \arctan2(r_2\sin(\theta_2 - \theta_1), r_1 + r_2\cos(\theta_2 - \theta_1))$$
References
