Computer Programming with Java

Course Pack for
ICS4U1 and ICS4UE

By G. Ridout

Version 1.1
© August 2014

Table of Contents

Preface

The material presented in this text was originally used to teach additional topics in the Java programming language to Grade 12 Computer Science students. It assumes students have experience in the Java programming language.

This text makes no attempt to cover all of the material in the Grade 12 Computer Science course. Since this is the first version of the printed course pack it does not cover all of the programming concepts that will presented in the course. More topics will be covered in later versions. It also does not cover topics such as Environmental Stewardship, Ethical Issues, Emerging Technologies and Society and Exploring Computer Science. Finally, it does not cover Java Swing Applications or Android Apps, although these topics may be presented in the grade 12 course to help students with their final projects.

Since this text is a work in progress any corrections, comments or suggestions would be greatly appreciated.

Finally I would like to thank all of the students who I have taught and colleagues I have worked with at Richmond Hill High School who have given me valuable feedback, suggestions and comments on the material presented.

G. Ridout
August 2014

Chapter 1 – Review of Java and Problem Solving

1.1 Grade 11 Course Pack Review

This course pack assumes that you have already learned the basic Java programming topics covered in the Grade 11 course pack "Introduction to Programming with Java". Since some of your knowledge may need refreshing, here are some of topics that you should review. Once you have reviewed these topics, you should also complete Questions 1 to 14 at the end of this chapter.

Eclipse IDE

You should start by installing the latest version of Java (currently Java 8) as well as the latest "standard" version of the Eclipse IDE (currently 4.4). If you did not use Eclipse in Grade 11 or you need a refresher, you should look over Section 1.5 in the Grade 11 course pack. Also see Appendix B in the Grade 11 course pack for tips on finding and correcting errors. Finally, see Appendix C.3 in this course pack for more tips on using Eclipse.

Java Compiler and the JVM

You should know how Java programs are converted to machine code, so take a few minutes to look over Section 1.3 in the Grade 11 course pack.

Java Style Guide

You should know how to name identifiers and how and when to comment your Java programs. Review Appendix A in this course pack.

Basic Java Data Types

You should know about the 5 basic primitive types in Java (boolean, char, int, long and double) including casting and promotion as well as Strings. Review sections Section 2.1 and 2.2 in the Grade 11 course pack.

Basic Input and Output

You should know how to use `Scanners` for keyboard input and the `println` and `printf` methods for program output. Review sections Section 2.3 and 2.4 in the Grade 11 course pack. If you used the Ready `Console` class for input and output in Grade 11, you will need to read over these sections carefully.

Arithmetic Expressions and Math and Character Methods

You should know all of the operators including % and integer division as well as how all of the Math and Character class methods work such as `Math.round`. Review sections Section 2.5, 2.6, 2.7 and Appendix D in the Grade 11 course pack.

Loops and Decisions

You should know how to make decisions using `if` statements including how to use `else` and `else if` statements to make your code more efficient. You should also know how to use conditional loops (`while` loops – don't worry about `do while` loops) and counting loops (`for` loops). Review Chapter 3 in the Grade 11 course pack.

Java Methods

You should know the advantages of using methods in Java and how to write methods in Java. You should also know how to write proper Java doc comments for methods including the `@param` and `@return` tags. Review Chapter 4 in the Grade 11 course pack and Appendix A is this course pack

String Manipulation

You should know the methods used to manipulate Strings in Java including `length`, `charAt`, `substring`, `indexOf` etc. Review Chapter 5 in the Grade 11 course pack. If you haven't used Scanners before you should review Section 5.8 on how to use Scanners to break up a String.

Arrays in Java

You should know how to create and use both 1D and 2D arrays in your Java programs. Review Chapter 7 in the Grade 11 course pack.

Reading Data from a Text File

For some of the practice problems at the end of the chapter, you will need to be able to read data from a text file. Look over Appendix E.1 in this course pack.

1.2 Software Development Life Cycle

Whether you are solving a simple problem or creating a more complicated piece of software there are some basic steps you need to follow when developing your solution. Although the steps are listed in sequence, in many cases there will be overlap and backtracking between each of the following steps. Even though <u>final</u> testing is listed as the second last step, it is important to test your ideas and code at each step as you work towards your final solution. Finding and correcting errors in both your design and code at the earliest possible stage is very important.

1) Define the Problem

- Read over the problem carefully and make sure that you understand what you are being asked to do. Ask questions if the problem is not clearly understood.
- For larger programs you will need to define the scope of the program identifying which features that you plan to include.
- This is a very important first step since you can't possibly find the correct solution unless you have identified the problem correctly.

2) Analyze the Problem (Initial Design)

- Identify the user requirements (outputs) and what information you are given (inputs). For more complicated problems you may want to create a sample input/output or detailed screen layouts.
- Think about the memory and data structures (objects, arrays etc.) that you will need to solve the problem. Look at both the inputs and outputs for ideas.
- Identify the main steps that will be needed to solve this problem. For larger problems you many want to break the problem down into smaller sub-problems (methods) and/or objects.
- If you need to create some new objects/classes, you should identify the data (characteristics) and methods (behaviours) required for each new object.

3) Design the Solution (More Detailed Design)

- Plan out the required steps (algorithms) for each method in the main program or any new objects.
- Create a <u>comprehensive</u> set of test data that can be used to test both your algorithm and your code. (See 1.3 for more on testing). For many problems, working through and solving the test cases manually will help you to come up with a better algorithm to solve the problem.
- Carefully check that the design produces correct results in an efficient way.
- Use flowcharts or pseudo code to help clarify your algorithms.

4) Implement the Solution

- Implement (code) the solution using the most appropriate tool (e.g. Java)
- If possible, use built in tools (e.g. objects and methods) to simplify your solution.
- Find and correct any errors including compiler/syntax and logic errors
- Code should adhere to a defined programming style and should include internal documentation (comments) (see Appendix A).

5) Final Testing and Debugging

- Find and correct logic errors and run-time errors
- Use the comprehensive set of test data created earlier to check that the final program is a robust and complete solution to the original problem.
- Finding and correcting errors at this stage can be more difficult so it is very important that you test your design and code as it is being developed so that you correct any errors as soon as possible.

6) Deployment and Maintenance

- Deliver the final product to the user (hand in your final program).
- Re-visit the above steps as required as specifications change or new problems arise.

For the first part of the course most of our assignments will be smaller programs but towards the end of the course we will be looking at bigger projects including the final project. When we start working on bigger projects we will re-visit and expand our approach to the Software Development Process and look at other ideas such as Object Oriented Design.

1.3 Testing your Algorithms and Code

When developing solutions to problems, finding correct solutions that work in all cases is your top priority. Finding a solution that only works most of the time is not acceptable. In order to check both your initial algorithm and your code it is important to come up with a comprehensive set of test cases. Here are some guidelines to help develop good test data:

1) For numerical input consider boundary cases (small, large, left, right) and cases on the edge of the boundaries (look at cases +/- 1 from the boundary cases). You should also consider special cases such as 0 and 1. To help come up with these cases, carefully look at the problem specifications.
2) For string input consider empty strings, the longest possible string and all types of valid characters.
3) Re-read the problem carefully to help identify any special test cases. Hunt for weaknesses e.g. "always" and other absolutes. Don't just consider the given test cases since in many cases these are only simple test cases to get you started.
4) Consider how multiple possible solutions are handled especially if you are looking for an optimum solution. Develop test cases to check these situations.
5) For small non-trivial problems you may want to consider all possible cases. If possible, write a program to help generate your test cases.
6) Test the efficiency of your code by including large or complicated test cases that may take a long time to calculate the answer.
7) Looking at your code, make sure that your test cases give complete "code coverage" (every section of your code should be reached at least once). This is especially important for any decision or loop structures including any boundary conditions for any comparisons.

To prove that an algorithm is correct without testing all possible test cases may sometimes be difficult. However, all it takes is one good counter example to prove an algorithm is incorrect. Good counter examples can limit the amount of time you waste on an algorithm that won't work. A small and simple counter example that disproves an algorithm is usually easier to check and more eloquent.

Question 16 in the questions at the end of the chapter looks at some more ideas on Software Testing.

1.4 Writing Simple Code

In addition to writing code that is correct, you also want to write simple code that is easy to follow. Reasons for writing simple and easy to follow code include:

1) Simple code is easier to review and comment. If you are having trouble commenting your code because it is too complicated to explain, maybe you should try to simplify your code.
2) Simple code is easier to check. White box testing of code with complicated "if" structures is a lot harder than checking code with simpler "if" structures.
3) Simple code is easier to update or modify. If you need to update your code, simple code is easier to work with. This is especially true if you are modifying code that was written a few months ago or by someone else.
4) Last but not least, simple code is easier for your teacher to mark.

Here are some tips on how to write simple code.

1) Think carefully before writing any code. A simple, easy to follow algorithm is the first step to writing simple easy to follow code.
2) Break your code down into methods or objects. Since methods perform a specific well defined task, the code will be easier to follow. Grouping your code into objects can also help.
3) Write the comments for your code as you go. As was mentioned earlier, if your comments are hard to write or too complicated, your code may need to be simplified.
4) Review your code as you are writing it to make sure that it is easy to follow and not over complicated.
5) After adding code to handle special cases, step back and see if your code can be simplified by eliminating any redundant code. You may not need all of the checks that you thought you needed.
6) Have someone else review your code. If he/she can't understand your code it probably needs to be simplified. Knowing someone is going to be looking at your code helps to motivate you to write cleaner code. Also, you may find that during the process of explaining your code to someone else, you will come up with some ideas to improve your code.
7) Be careful when cutting and pasting large sections of code. This can lead to duplicate sections of code that are harder to follow.

Even though your first objective is always to write code that solves the problem, producing correct results for all test cases, writing simple and easy to follow code is also very important. If you follow the guidelines given above you will start to develop better habits that will lead to writing cleaner code.

1.5 Writing Efficient Code

In addition to correct, simple and easy to follow code, you also want to write code that is efficient. Efficient code should run quickly and not use up too much memory. Sometimes there are tradeoffs between simplicity, speed and memory usage but in many cases, well written code will optimize all three properties.

Generally, to make your code run faster your objective is to write code that minimizes the amount of work the computer needs to do. Here are some suggestions:

1) Carefully check any loops in your code. Since code inside loops is run multiple times, this is the first thing you should look at when trying to make your code run faster. For each loop you need to consider if the loop is necessary and if it is necessary, can the number of times that the loop repeats be reduced. For example, to speed up the checking loop in the `isPrime` method, if we deal with all even numbers up front we only need to check for odd factors. This will cut the number of checks in the loop in half.
2) You should also look at what is being done in each loop. Is all of the code inside the loop necessary? In some cases you may be able to move some of the code outside of the loop so it doesn't need to be repeated.
3) Check for any redundant or unnecessary code in your program. If speed is an issue, you should review your code to see if all of the given code is needed based on the problem and potential inputs to the program.
4) Look at the program specifications when picking variable types. Smaller memory locations take up less space and require less processing time during calculations and comparisons. If you can use an `int` instead of a `long` or a `double`, your code will be faster and use less memory. Using a `short` or another smaller integer instead of an `int` may save some memory but it won't save any time since `shorts` are converted to `ints` for calculations in Java.
5) Understand how built in methods such as `Math.pow` work. In some cases, writing your own code may be more efficient. This is especially important when this code is inside of a loop.
6) Understand how operations with different objects work. We will see in Chapter 2 that using a `StringBuilder` object is faster than using a `String` object for most operations.
7) Know the run time of different algorithms. For example, when searching a list a binary search is a lot faster than a linear search.
8) Have someone else look at your code. Sometimes we can't see problems with our own code. Having another set of eyes look at your code can help highlight possible ways to simplify and speed up your code.

For some smaller problems writing faster more efficient code may not be necessary. However, for larger more complicated problems inefficient code may take too long to find a solution. Also, some larger problems may run out of memory if the wrong data types are used or the memory is not managed properly.

1.6 Questions, Exercises and Problems

1) Draw a neat diagram to show what happens when you compile and run a Java program. Your diagram should include the following terms: `.class` files, `.java` files, bytecode, machine code, source code, Java Virtual Machine and Java compiler.

2) Complete the following chart:

Expression	Result	Type of Result
`15 % 8 + 13/7`		
`3+4 * 9-6`		
`(4.5 + 5.5 - 4) / (9 - 7)`		
`Math.round(5.521)`		
`Math.abs(-7) + Math.abs(4)`		
`Math.sqrt(16)`		
`Math.pow(3, 4)`		
`Math.max(Math.min(3.4, -4),9)`		
`Character.toUpperCase('g')`		
`Character.isDigit('K')`		
`7 >= 5 || 124 < 18`		
`!(23 == 32) && (12 != 99)`		

3) Given the following section of Java code, complete the following table indicating the predicted output and the number of comparisons required for each of the given test cases. Be careful.

```
if (first > 5 && first < 10)
   System.out.println("Apple");
else if (second < 0 || second >= 12)
   System.out.println("Banana");
else if (third < second && third < first)
   System.out.println("Grape");
else
   System.out.println("Peach");
```

Case	first	second	third	Output	No. of Comparisons
1	7	-4	-12		
2	17	-4	15		
3	0	9	12		
4	-10	13	31		
5	21	1	0		
6	2	4	3		

4) Write the Java code to generate a random integer number between -5 and +5 inclusive (including both -5 and +5) using the `Math.random` method.

5 a) Predict the output of the following section of Java code: Output

```
int total = 0;
for (int number = 3; number < 8; number++)
{
   total += number;
}
System.out.println(total);
```

b) Re-write the above section of code using a while loop:

6) Predict the exact output of the following section of Java code.

```
for (char last = 'E'; last >= 'A'; last-=2)
{
   for (char ch = 'A'; ch <= last; ch++)
      System.out.print(ch);

   System.out.println();
}
```

7) Given the following method:

```
static int mystery(int first, int second)
{
     int remainder = first % second;
     while (remainder != 0)
     {
         first = second;
         second = remainder;
         remainder = first % second;
     }
     return second;
}
```

a) Predict the return value of the following method calls. In each case, you may want to complete a memory trace of the local variables in the `mystery` method.
   i) `mystery(36, 24)` ii) `mystery(7, 21)` iii) `mystery(14, 5)`

b) Explain what the `mystery` method does.

8) Predict the exact output of the following sections of code:

```
int first = 9876;
double second = 87.248;
String name = "Bugs Bunny";
System.out.printf("First:%,6d Second: %.1f%%%n%-15sXX", first, second, name);
```


```
for (int number = 10; number <= 1000; number *= 10)
{
   int square = number * number;
   int cube = square * number;
   System.out.printf("%06d%8d%11d%n", number, square, cube);
}
```


9) Assume that you want to write a program which reads in a series of names and marks and then displays the name and the mark of the student with the highest mark. Fill in the blanks to complete the following code:

```
String highestStudent = null;
int highestMark = ________________;
Scanner keyboard = ____________________________________________;
System.out.print("Enter the number of students: ");
int noOfStudents = _________________________________________;
_____________________________________________________________;
for (int student = 1; student _______________________; student++)
{
   System.out.print("Please enter the next name: ");
   String name = ________________________________________;
   System.________________________________________________________;
   int mark = ____________________________________________;
   if (_________________________________________________)
   {
      ______________________________________________________;
      ______________________________________________________;
   }
   ___________________________________________________________;
}
System.out.println(____________________________________________
      ______________________________________________________________);
```

10) Find and correct all of the errors in the following Java methods. In each case indicate whether the error is a compiler, logic or run-time error. Use the method comments and heading to figure out what each method should be doing. You should also fix any problems that will make the code less efficient.

```
/** Finds the average of a collection of integers, given
 *  the total of the numbers and the number of numbers
 *  @param total the total of the numbers
 *  @param noOfNumbers the number of numbers
 *  Precondition: noOfNumbers >= 1
 *  @return the average of the numbers
 */
public static double average(int total, int noOfNumbers)
 {
     double average = total / noOfNumbers
 }

/** Finds the index of the largest element in an array of integers
  * @param list the array of integers
  * Precondition: list contains at least one element
  * @return the index of the element with the largest value
  */
 public static void indexOfLargest(int [] list)
 {
     int indexOfLargest = 0;
     for (int index = 0 ; index < length ; index++)
     {
         if (index > indexOfLargest)
             indexOfLargest = index;
     }
     return index;
 }

/** Finds the sum of all of the factors of a number
  * @param number the number to find the sum of the factors for
  * Precondition: the given number is positive (> 0)
  * @return the sum of the factors of the given number
  */
 public static int sumOfFactors(int number)
 {
     int sumOfFactors = 0;
     for (int check = 1 ; check < number ; check ++)
     {
         if (check % number == 0)
             sumOfFactors = check;
     }
      return sumOfFactors;
  }
```

11) Given the following Strings:

```
//                              1         2         3         4
//                    01234567890123456789012345678901234567890
String firstStr  = "Welcome back to Computer Science!";
String secondStr = "abcdefghijklmnopqrstuvwxyz";
String thirdStr  = "apple peach apple pear apple pineapple";
```

Complete the following chart (Be careful to make sure you are using the correct string):

Method Call	Return Value
`firstStr.length ()`	
`firstStr.charAt (18)`	
`firstStr.substring(25)`	
`secondStr.substring (17, 19)`	
`secondStr.indexOf('L')`	
`thirdStr.indexOf ("apple", 14)`	
`thirdStr.lastIndexOf ("apple")`	
`thirdStr.lastIndexOf ("apple", 24)`	

12) Given the following methods, predict the return values of the method calls given below:

```
static int one(String str, char ch)
{
   for (int index =0; index < str.length(); index++)
      if (str.charAt(index) == ch)
         return index;

   return -1;
}
```

a) `one("Welcome to ICS4U",'o')`

b) `one("Review Question",'S')`

```
static String two(String str)
{
   String newStr = "";
   int index = str.indexOf(' ');
   while (index > 0)
   {
      newStr = str.substring(0,index) + newStr;
      str = str.substring(index+1);
      index = str.indexOf(' ');
   }
   return str + newStr;
}
```

c) `two("one two three")`

d) `two("ne do ll we")`

13) Write the complete Java code (including full comments with introductory javadoc comments) for a method called `remove` that removes an element from an array of Strings. The method will have two parameters, the array to remove the element from and the index of the element you want to remove. It should return a new array with the given element removed. For example, given the following:

```
String [] origList = {"one", "two", "three", "four", "five"};
String [] newList = remove(origList, 2);
```

The value of `newList` will be: `{"one", "two", "four", "five"}`

You can assume that the array will contain at least one element and that the remove index will be valid (0 <= remove index < array length). To help get you started, the method heading is given below:

```
public static String [] remove (String [] list, int removeIndex)
```

14) Complete the code for a method called `insertElementAt()` that inserts a number into an array at a specified index shifting the elements in the array as required. This method has 3 parameters, the array to insert into, the index of the inserted number and the number to insert. It should return a new array with the number inserted. For example, after running the following code:

```
    int [] originalArray = {30, 17, 79, 65, 98};
    int [] newArray = insertElementAt (originalArray, 3, 73};
```

`newArray` should be `{30, 17, 79, `**`73`**`, 65, 98}`. You can assume that the `insertIndex` will be valid (0 <= insertIndex <= length of array). Don't forget to include comments to explain each section of code. To help get you started, the method heading is given below:

```
static int [] insertElementAt (int [] list, int insertIndex,
                                        int numberToInsert)
```

15) Create a test plan for an `isPrime` method that checks to see if a given `int` number is a prime number or not. Your test plan should include the following headings:

Case #	Number	Output (true/false)	Reason for including this case	OK ✓

Include at least 10 test cases in your plan. Consider minimum and maximum values as well as special cases that may be missed in an `isPrime` method. To test your test cases, you will be presented with different versions of the `isPrime` method. Your test cases should catch any problems with these methods including any inefficient code.

16) Write a simple Java program that produces an inventory report based on the data in a text file. The given file will contain the product name, price and quantity on hand for one or more products. A sample input file is shown on the right. For each product your program should display the inputted information as well as the total value of each product (quantity × price). Your program should also display the total value of all the products entered as well as the product with the highest total value – display both the name of the product and its value. Use the `printf` method to properly line up the data in nice columns to match the sample given below:

Sample input file:
```
Mouse Pads
4.67
10
HDMI Cables
10.54
12
Phone cases
19.89
5
```

```
                     Inventory Report

Product Name           Price ($)   Quantity   Total Value($)
Mouse Pads                4.67          10          46.70
HDMI Cables              10.54          12         126.48
Phone cases              19.89           5          99.45
Overall total                                      272.63

The highest total value for an item is $126.48 for HDMI Cables.
Thank you for using the Inventory Management Program
```

17) To find the digital root of a number, you start by finding the sum of the individual digits in the number. If this sum has more than 1 digit, you repeat the process to find the sum of the digits in the sum. This process is repeated until the final sum is one digit. This final digit is the digital root of the number. For example, if the original number was 1765456789, the first sum of the digits would be 1+7+6+5+4+5+6+7+8+9=58, the second sum would be 5+8=13 and the final sum would be 1+3=4. Therefore the digital root of this number would be 4.

Write a Java method called `digitalRoot()` that finds and returns the digital root of a positive integer number (use a `long` number). Also write a main program (see sample below) to test your method. You can quit your main program by closing the console window.

```
                        Finding Digit Roots
Please enter a number: 54321
The digit root of 54321 is 6

Please enter a number: 9223372036854775837
The digit root of 9223372036854775837 is 1
```

Hint: You can use % and / to break a number down into its individual digits.
For example, given number = 12345, number%10 is 5 and number/10 is 1234
As a first step, find the sum of the number's digits.

18) Write a Java program to solve the quadratic equation: $ax^2 + bx + c = 0$ where the values of a, b and c are given by the user. For each set of inputs your program should describe the type of roots and give the value of any roots (rounded to 2 decimal places). Your program should be able to handle all possible real values for a, b and c. If $a = 0$, the equation simplifies to the linear equation $bx + c = 0$. If $a \neq 0$, you can use the quadratic formula given on the right.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The following is the start of a test plan for your program. Your first step is to complete this test plan to provide a set of test data for all 6 possible cases. When testing your actual program you may want to include additional test cases.

Test Plan – Input/Output Analysis

Case	a	b	c	Expected Output	✓
1	0			Every real number is a root.	
2	0			There is no solution.	
3	0			The equation has one real root: _________	
4				The equation has no real roots	
5				The equation has one real root: _________	
6				The equation has two real roots: ______ and ______	

Before starting on the Java code, you should carefully plan out the logic for your program on paper, listing all of the steps required to complete this program. You should minimize the number of comparisons required to deal with each possible test case. Don't forget to test your algorithm using the test plan. After completing and testing your algorithm you should write and test your Java code.

19) Given an integer number greater than 1, re-write this number as a product of its prime factors. You should also count and display the number of prime factors as well as the run-time (in milliseconds) for each test case. Here is a sample run with a couple of test cases. You should come up with some more cases to test your code.

```
                   Finding Prime Factors

Please enter a number: 120
120 = 2×2×2×3×5
Number of prime factors: 5
Run time: 0.2 ms

Please enter a number: 223092870
223092870 = 2×3×5×7×11×13×17×19×23
Number of prime factors: 9
Run time: 0.3 ms
```

Note: The Unicode for × is `00D7` – Use `\u00D7` in your format string.

20) For the following problem you want to write both simple and efficient code, so it is important that you plan out your solution very carefully <u>before</u> starting on the Java code.

Given a list of integers, find the largest sum of consecutive numbers in the list. For example, for the following list of integers:

```
          4   5   -11   6   8   -12   4   5   -2   6   -10   3   -4
```

the largest sum of consecutive numbers is 15 (6 + 8 + -12 + 4 + 5 + -2 + 6).

To test your program you will be given a data file with the following information:

- The first line of the file will contain the number of test cases.
- Each test case will contain 2 lines: the first line will indicate how many integers are included in this case (maximum of 1,000,000), and the next line will contain the integers with a single space separating each number. Each integer will be between -1000 and 1000 inclusive.

Here is a sample input file:

```
2
13
4 5 -11 7 8 -12 3 5 -2 6 -10 3 -4
7
-3 -5 -7 2 -4 -2 1
```

Your program should output to the screen the largest sum of consecutive numbers for each test case. Output for the above input file would be:

```
Largest Sum of Consecutive Numbers
Case #1: 15
Case #2: 2
End of output
```

Remember you will need to plan out your solution carefully before starting on the code since your program needs to be efficient in order to handle list sizes of up to 1,000,000.

21) If you multiply certain integer numbers by 3, the resulting product is made up of the same digits as the original number. For example, if you triple the number 24714 you get 74142 which contains the same 5 digits ('2', '4', '7', '1' and '4') as 24714. Write a Java program to find all 5 digit numbers that satisfy this property. Note: 01234 is not a 5 digit number. Using the same program, find out <u>how many 9 digit numbers</u> satisfy this property.

Hint: You can use % and / to break a number down into its individual digits.
For example, given `number = 12345`, `number%10` is 5 and `number/10` is `1234`

Note: To get the answer to the 9 digit numbers problem in a reasonable time (less than 10 seconds), your program will need to be efficient.

22) In section 1.3 we looked at some ideas to help test your algorithms and/or code. To gain more insight into bigger picture software testing, do some research on-line and answer the following questions:

a) The 4 main stages/levels of testing include Unit Testing, Integration Testing, System Testing and Acceptance Testing. Describe each of these stages of testing giving a specific example of what you would do at each stage.
b) What is the difference between White Box and Black Box Testing? Give a specific example of each type of testing. At which stage would you perform White Box testing? At which stage would you perform Black Box testing?
c) What is Regression Testing? At which stage would you perform regression testing?
d) What are Alpha and Beta Testing? At which stage would you perform alpha and beta testing?

23) Re-write each of the following sections of code to make each more efficient. The new code should produce the exact same results as the original code. Each section of code is separate.

```
for (int product = 0; product < price.length; product++)
   price[product] = cost[product] * oldMarkUp * increase;
```

```
for (int index = 0; index < numbers.length; index ++)
{
   if (index % 2 == 1)
      numbers[index] = 1;
   else
      numbers[index] = 2;
}
```

```
for (int number = 0; number < cubes.length; number++)
{
      cubes [number] = (int)Math.pow(number, 3);
}
```

Chapter 2 – StringBuilders and ArrayLists

In Grade 11 we used `Strings` to store strings of characters and arrays to store lists of various types of data. Both of these data types have their limitations. In this chapter we introduce two new classes: `StringBuilders` and `ArrayLists` that have some advantages over the `Strings` and arrays used in Grade 11.

2.1 `Strings` vs. `StringBuilders`

As you may recall `String` objects in Java are immutable which means they cannot be changed. We can change a `String` reference to refer to a new `String` object but we can't change the contents of a `String` object. If we need to manipulate the contents of a string, a more efficient option is to use a mutable `StringBuilder` object. Here are two examples to show the differences between `Strings` and `StringBuilders`.

Capitalizing the First Character in a String

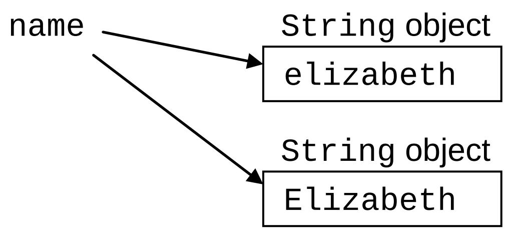
Assume that we want to capitalize the first letter in a `String`. Given a string stored in a `String` variable called `name`:

```
String name = "elizabeth";
```

We could capitalize the first letter (i.e. change `"elizabeth" to "Elizabeth"`) using the following code:

```
name = Character.toUpperCase(name.charAt(0)) + name.substring(1);
```

Although this code seems quite simple, if we look at what is happening in memory, this code is not very efficient.



Once `name` is reassigned to the new `String`, the old `String` object may be garbage collected.

Since `String` objects are immutable we can't just change the first letter from 'e' to 'E'. Instead we need to complete the following steps:

1) Create a new `String` object to hold the updated name.
2) Put a capital first letter into this new `String` object.
3) Copy over the rest of the letters from the original `String` into the new `String`.
4) Change the `String` reference `name` to refer to the new `String` object.
5) Garbage collect the original `String` object.

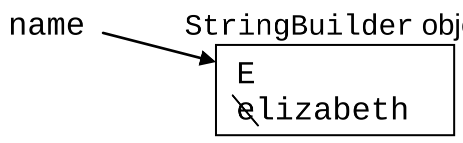
Alternately, we can use a `StringBuilder`. A `StringBuilder` is a mutable object that has extra methods that can be used to manipulate its contents. Therefore, we can capitalize the first letter in a `StringBuilder` more efficiently. Like most objects (except for `Strings`), we need to use the `new` command to construct the new `StringBuilder` object. For example:

```
StringBuilder name = new StringBuilder("elizabeth");
```

Then to capitalize the first letter (i.e. change `"elizabeth"` to `"Elizabeth"`) we use the following code:

```
name.setCharAt(0, Character.toUpperCase(name.charAt(0)));
```

This code uses the `setCharAt` method that allows us to set that value of individual characters in a `StringBuilder`. It has 2 parameters, the index of the character that we want to change and the new character that we want to set this character to. In the above example, we are setting the first character to the uppercase value of the original character. Since `StringBuilder` objects are mutable this is allowed. This code is a lot faster than the earlier code since we don't need to create any new objects, copy over any characters, change any references or garbage collect any old objects. Here is what happens in memory with a `StringBuilder`.



Since `StringBuilder` objects are mutable, we can change characters in a `StringBuilder` directly.

In addition to many `String` methods such as `charAt`, `StringBuilders` also include methods that can be used to manipulate their string contents. See Appendix D for a complete list of `StringBuilder` methods and section 2.2 for some examples.

Appending Characters to the end of a String

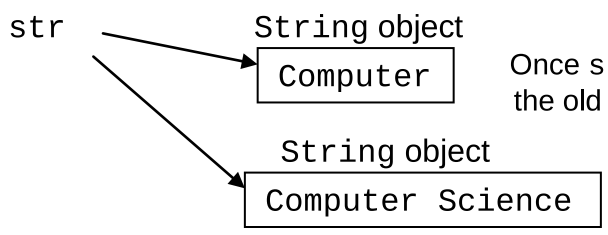
Assume that we want to add some extra characters to the end of a `String`. Given that we have a string stored in a `String` variable called `str`:

```
String str = "Computer";
```

We could add some new text to this `String` using the += command. For example:

`str += " Science";` will change `str` to refer to `"Computer Science"`

Again, although this code seems quite simple, if we look at what is happening in memory, this code is not very efficient.



Once `str` is reassigned to the new `String`, the old `String` object may be garbage collected.

We can't just add the new text to the end of the `String`. Once again, we need to:

1) Create a new `String` object.
2) Copy over the original letters `"Computer"` to the new `String` object.
3) Add in the new letters `" Science"`.
4) Change `str` to refer to the new `String` object.
5) Garbage collect the original `String` object.

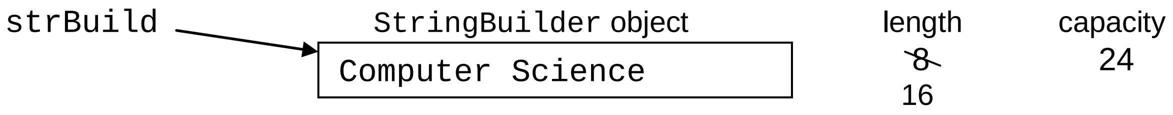
Using a `StringBuilder` we can do the same thing more efficiently. Not only are `StringBuilders` mutable, they also include extra capacity to make room for any new characters that may be added. Given that we have a string stored in a `StringBuilder` variable called `strBuild`.

```
StringBuilder strBuild = new StringBuilder("Computer");
```

To add some new text to a `StringBuilder` we use the `append` command.

```
strBuild.append(" Science");
```

When we created the original `StringBuilder` object, in addition to the space needed to store the 8 characters in `"Computer"`, space was also reserved for 16 extra characters to give a total capacity of 24 characters. Since `" Science"` fits within this capacity and since `StringBuilder` are mutable, we can easily add in these extra letters without all of the extra steps given earlier for the `String` object. A `StringBuilder` keeps track of both its length (actual number of characters) and its capacity (amount of space available).



This code is faster than the earlier code since we don't need to create a new object, copy over any characters, change the reference and garbage collect the String.

If we added more characters to this `StringBuilder`, at some point its new length may become greater than its capacity. In this case, a new `StringBuilder` is automatically created with a higher capacity. The actual value of the new capacity is:

new capacity = 2 × (old capacity + 1)

In this case, when the new `StringBuilder` is created, steps similar to the steps required when creating a new `String` are carried out. This includes: creating a new `StringBuilder`, copying over all of the characters, changing the `StringBuilder` reference and garbage collecting the original `StringBuilder`. Therefore, if possible, we should try to set the initial capacity of a new `StringBuilder` to the anticipated size of the final string to avoid this extra work later on. This is especially important when we are appending a lot of characters to a `StringBuilder`.

2.2 StringBuilder Methods and Examples

In addition to the setCharAt and append method shown in the previous sections, other StringBuilder methods can be found in Appendix D. Here are some tips on using these methods.

When you construct a StringBuilder you have 3 options. You can construct an empty StringBuilder with an initial default capacity of 16. If you know the final length of your StringBuilder, you can also construct an empty StringBuilder with a set capacity. Finally, you can construct a StringBuilder with the initial contents of a given String. The capacity in this case will be the length of the original String plus 16 extra characters. Here are some examples.

```
StringBuilder emptyWithDefaultCapacityOf16 = new StringBuilder ();
StringBuilder emptyWithSetCapacityOf10 = new StringBuilder (10);
StringBuilder startWithCapacityOf21 = new StringBuilder ("Start");
```

When using setCharAt it is important to remember that the index in this method must be a valid index <u>based on the length</u> of the StringBuilder <u>not</u> its capacity. The setCharAt can only be used to change the value of a character that already exists in the StringBuilder so 0 <= index < length. For example:

```
StringBuilder strBuild = new StringBuilder("Comic");
strBuild.setCharAt(5, 's');
```

will throw a StringIndexOutOBoundsException since 5 is out of range even though the capacity of the StringBuilder has room to add the 's'. To add characters to the end you need to use append not setCharAt.

The deleteCharAt method deletes a single character, shifting all of the character after this character one to the left. It will decrease the length of the StringBuilder by 1.

```
StringBuilder strBuild = new StringBuilder("ABCDEFG");
strBuild.deleteCharAt(3);
System.out.println(strBuild);
```

would display ABCEFG since it will delete the character at index 3.

If you want to delete more than one character you can use the delete method. This method has 2 parameters to indicate the start (inclusive) and end (exclusive) of the section that you want to delete. The number of deleted characters will be end – start.

```
StringBuilder strBuild = new StringBuilder("ABCDEFG");
strBuild.delete(1, 3);
System.out.println(strBuild);
```

would display ADEFG since it will delete the characters at indexes 1 and 2 (not 3).

To insert an object including another `String` into a `StringBuilder` you can use the `insert` method. The object to insert is given as the second parameter and the index of the inserted object in the new `StringBuilder` is the first parameter. All characters to the right of the insert index will be shifted to the right to make room for the inserted object. For example:

```
StringBuilder strBuild = new StringBuilder("ABCDEFG");
strBuild.insert(2, "XYZ");
System.out.println(strBuild);
```

would display `ABXYZCDEFG` since it will insert `"XYZ"` with the index of the `X` being 2.

A combination of `delete` and `insert` is the `replace` method that replaces a section of the `StringBuilder` with another set of characters. For example:

```
StringBuilder strBuild = new StringBuilder("ABCDEFG");
strBuild.replace(1, 3, "XYZ");
System.out.println(strBuild);
```

would display `AXYZDEFG` since it will delete `"BC"` and insert `"XYZ"` in its place.

Other methods that you may be interested in are `capacity` that tells you the current capacity of the `StringBuilder` and `ensureCapacity` that can be used to set the capacity to a minimum value. The actual capacity will be determined by the resizing method given earlier (new capacity = 2 × (old capacity + 1)). For example:

```
StringBuilder strBuild = new StringBuilder();
strBuild.ensureCapacity(20);
System.out.println(strBuild.capacity());
```

would display `34` since to get a capacity of at least 20, we take 2 × (16+1). Normally we should set the capacity when we construct the `StringBuilder` but, if needed, `ensureCapacity` can be used to set the capacity later in the program.

You may have noticed that many of the `StringBuilder` methods return a `StringBuilder`. This is just a reference to the `StringBuilder` that we are working with. In most cases this return value will be ignored but it can be used in some cases to save a line of code. For example, if you have a method that returns a `String` and you want to remove the last character from a `StringBuilder` that you have been using before returning the result, you could use:

```
return strBuild.deleteCharAt(strBuild.length() -1).toString();
```

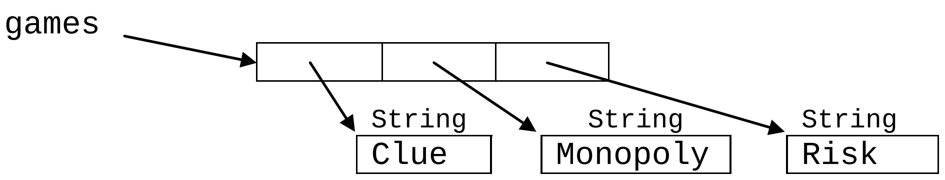
In addition to the many methods given in Appendix D, you can also use most of the `String` methods with `StringBuilders`. For example: `charAt()`, `indexOf()`, `lastIndexOf()` , `length()`, and `substring()` . However, the `equals()` method <u>does not</u> compare the contents of two `StringBuilders`. To compare two `StringBuilders` use `toString` and then compare the resulting `Strings`. Also, the `compareTo` method is not available so once again you will need to convert the value to a `String` (using `toString`) first.

2.3 Arrays vs. `ArrayLists`

When you create a new array object such as:

```
String [] games = {"Clue", "Monopoly", "Risk"};
```

A detailed memory diagram for this array shows the following:



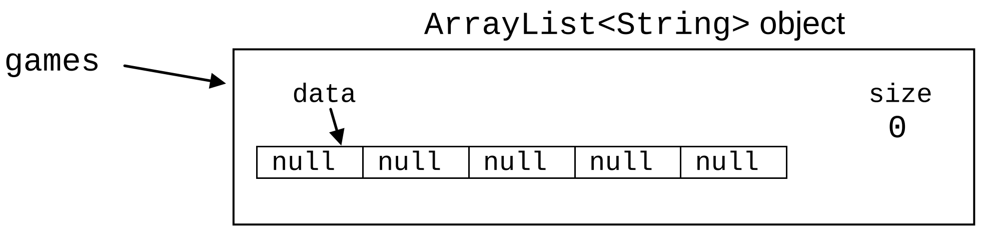
In this case we have an array of `String` object references. The array reference `game` keeps track of the position of the first element in the array. In this case, the `String` references in the array are physically beside each other in memory. This makes it easy to access individual elements in the array since only a simple calculation is needed to find an individual element's address. For example, to find the memory address of `games[2]`the JVM just takes the address of the first element given by `games` and adds the index (e.g. 2) times the size of each array element reference. This is one of the reasons why arrays in Java are zero based. Having all of the elements of an array next to each other in memory makes this indexing process very fast. However, it can cause problems when we want to add or remove elements from an array.

Since array elements are physically beside each other in memory and the length of an array is fixed, if we want to add new elements to or remove elements from an array, we need to create a new array, copy over the original elements of the array and then change the array reference to the new array. Note: This is a similar process that is used to re-size a String. See also questions 13 and 14 at the end of Chapter 1.

To make it easier to re-size an array (insert or remove elements), Java introduced `ArrayList` objects. `ArrayList` objects include extra capacity which allows re-sizing without creating new objects. Let's create the list given above using an `ArrayList`.

```
ArrayList<String> games = new ArrayList<String>(5);
```

A detailed memory diagram for this `ArrayList` shows the following:



The `ArrayList` object contains an <u>array</u> of `String` references of size 5 (specified when constructing the `ArrayList`). It also has a `size` variable to keep track of the actual number of elements in the `List`. Just like `StringBuilders` this extra capacity is not available to us directly; it is just there to avoid extra copying of array elements when adding new elements to an `ArrayList`.

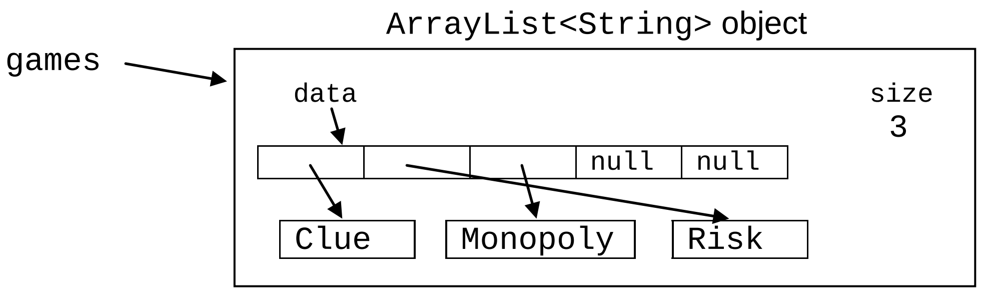
Since an `ArrayList` has a separate `size` variable we can add or remove elements without creating a new array. Also, if we add or remove an element, the other elements in the `ArrayList` are automatically shifted as required. To add elements to the end of an `ArrayList`, we can use the `add` method.

```
games.add("Clue");              // index will be 0
games.add("Monopoly");          // index will be 1
```

We can also add elements into the `ArrayList` at a specific index by adding another parameter to the `add` method. For example:

```
games.add(1, "Risk");           // index will be 1
```

Note: the index used with the `add` method must be valid based on the `size` of the `List` and not the capacity. There can be no gaps in our List. So in the last example, index would have to be 0, 1 or 2 since these are the only valid indexes for the third element. Here is what the `ArrayList` would look like in memory after adding the 3 elements. Notice that although `Risk` was added last, it was added at index 1, so `Monopoly` was automatically shifted over by changing the `String` references.



As elements are added to the List, the `size` variable is automatically adjusted. To access the size of the List, we use the `size()` method (instead of `length`). For example, to display all of the elements of the List we can use the following:

```
for (int i = 0; i < games.size(); i++)
    System.out.println(games.get(i));
```

Notice that we no longer use `[]` to access individual elements. Instead we need to use the `get` method. If we are looking at all of the elements of a `List`, an alternative loop structure is given below. This loop is sometimes called a “for each” loop since it looks at each element of the `List`. This “for each” loop also works with arrays.

```
for (String nextGame : games)
    System.out.println(nextGame);
```

Since we no longer use `[]` with `ArrayLists`, if you want to set individual elements in a List, you need to use the set method instead. For example:

```
games.set(0, "Jenga");    // similar to games[0] = "Jenga"
```

will replace `"Clue"` with `"Jenga"` at index `0`. It is important to remember that, even though we have extra capacity in an `ArrayList`, we cannot directly access these extra spots so `games.set(3, "Jenga")`would be invalid in the above case. However, since the `add` method adds to a `List`, using `games.add(3, "Jenga")` would be acceptable.

When adding new elements to an `ArrayList` if its size exceeds its capacity, the `ArrayList` will automatically create a new array in the `ArrayList` object with a larger size (one and a half times the original size), copy over the elements and update the references. This is similar to what happens with a `StringBuilder`.

Using an `ArrayList` adds the extra convenience of automatic resizing and taking care of shifting elements when adding or removing elements. There are also a few more methods (see the next section) that can make operations such as searching a list a little easier. However, there is a little more overhead (extra object, size variable, and extra capacity) with an `ArrayList` so if your lists have a fixed known length and you are not adding or removing elements you should probably still use an array. For lists with unknown lengths or if you are adding or removing elements to or from the list, an `ArrayList` is the better choice.

You should also note that in Java, `ArrayLists` can only contain objects (not primitives) so if you want an `ArrayLists` of `ints` you need to use the object equivalent `Integer` instead. With auto boxing and unboxing this is not usually a problem but an `ArrayList` of `Integer` objects will usually be slower than an array of `int` primitives. See the end of the next section for an example of an `ArrayList` of `Integer` objects.

2.4 `ArrayList` Methods and Examples

In the last section, we used two different versions of the `add` method to add elements to an `ArrayList`, the `get` method to access individual elements at a given index, the `set` method to set individual elements at a given index and the `size` method to find out the size of a `List`. We also saw how to use the “for each” loop to look at all elements in a List. Other `ArrayList` methods are given in Appendix D. Here is a demo to show how to use these methods. In each case, try to predict the output before looking at the result. Note: The results in this demo are cumulative.

```
// Create a new ArrayList of Strings with default capacity of 10
ArrayList <String> fruits = new ArrayList <String> ();

// Add some items to the list
fruits.add ("apple");
fruits.add ("peach");
fruits.add ("banana");
fruits.add ("kiwi");
```

In addition to the regular indexed loop and the for each loop, we can also display an `ArrayList` without using a loop. For example:

```
System.out.println(fruits);
```

would display: `[apple, peach, banana, kiwi]`

The `[ ]` are used to enclose the `List` and commas are added to separate each element in the List.

In the previous code we added elements to the end of the List. We can also insert elements in a specific location. For example:

```
fruits.add (1, "lemon");
System.out.println(fruits);
```

would display: `[apple, lemon, peach, banana, kiwi]`

If we only want to display part of the List, we can use the `subList` method. With `subList`, the first index is inclusive and the second index is exclusive. For example

```
System.out.println(fruits.subList(1, 4));
```

would display: `[lemon, peach, banana]`

Continuing with the same `List`, we can use `get` and `set` instead of `[]`

```
System.out.println (fruits.get (2));  // Like fruits[2]
fruits.set(2, "apple");               // fruits[2] = "apple"
System.out.println (fruits);
```

would display:

```
peach
[apple, lemon, apple, banana, kiwi]
```

In the above example, we ignored the value returned by the set method. The set method returns the element that is being replaced. So, the following code:

```
String removedFruit = fruits.set (3, "grape");
System.out.println (removedFruit);
System.out.println (fruits);
```

would display:

```
banana
[apple, lemon, apple, grape, kiwi]
```

Just like we can search `String` objects, we can search `ArrayList` objects using both `indexOf` and `lastIndexOf`. These methods use an element's `equals` methods to find matches. For example:

```
int index = fruits.indexOf ("apple");
System.out.println (index);
index = fruits.lastIndexOf ("apple");
System.out.println (index);
```

would display:

```
0
2
```

If we don't care about the position of an element in the `List` we can use the `contains` method to check if the element is in the `List`. For example:

```
System.out.println (fruits.contains ("lemon"));
System.out.println (fruits.contains ("peach"));
```

would display:

```
true
false
```

Removing elements from a list will automatically shift over the other elements and adjust the size. The `remove` method also returns the removed element. Like `set`, we can ignore this return value if we wish. For example:

```
removedFruit = fruits.remove (2);
System.out.println (removedFruit);
System.out.println(fruits);
```

would display:
```
apple
[apple, lemon, grape, kiwi]
```

Additional methods, such as `isEmpty` and `clear`, are described in Appendix D. Since `ArrayList` is part of Java's `Collections` framework, to sort an `ArrayList`, we can use the static method `Collections.sort`. For example:

```
Collections.sort(fruits);
```

It is important to remember that `ArrayList` are objects so if we want to create a copy of a `List` we need to actually make a new copy. For example, the code:

```
ArrayList <String> newList = fruits;
```

will not create a copy of the `List`. In this case, `newList` will just refer to the same `List` as `fruits`, so if we ran the command `newList.remove(0)` we would actually be changing the original `fruits List` as well. If we want to make a copy of the `List` we need to construct a new `List` using an `ArrayList` constructor.
For example:

```
ArrayList <String> newList = new ArrayList <String> (fruits);
```

will create a new `List` that is a copy of the original `fruits List`.

Since `ArrayLists` can only contain objects, here is an example of how to use an `ArrayList` of integers using auto boxing and unboxing.

```
List <Integer> intList = new ArrayList <Integer> ();
for (int number = 1 ; number <= 20 ; number++)
    intList.add (number);      // Ok since int is converted to an
System.out.println (intList);  // Integer object with “auto boxing”

int total = 0;
for (int number: intList)      // Ok since Integer object is converted
    total += number;           // back to an int with “auto unboxing”
System.out.println (total);
```

Without auto boxing and unboxing we would need to explicitly convert between an `int` primitive and an `Integer` object. For example, to add the `int` primitive to the list of `Integer` objects we need to change the line in the first for loop to:

```
intList.add (new Integer(number));
```

Then to get the `int` value to add to `total`, we would need to change the 2nd loop to:

```
for (Integer number: intList)
   total += number.intValue();
```

2.5 `Iterators` and `ListIterators`

When you want to iterate through a `List`, regular for loops and “for each” loops work well in most cases. One exception is when you want to remove elements from a List as you are iterating through it. This will not work with a “for each” loop (you will get a `ConcurrentModificationException`) and it can be tricky with a regular loop since your index value needs to be adjusted when you remove an element. To get around this problem we can use `Iterators` and `ListIterators`. `ListIterators` can also be used to go through a List in reverse order.

An `Iterator` is an object that can be used to transverse through all of the elements in a `Collection`. In Java, to look at elements in a `List` we can use a `ListIterator`. The following description is from the Java help for `ListIterator`.

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A `ListIterator` has no current element; its *cursor position* always lies between the element that would be returned by a call to `previous()` and the element that would be returned by a call to `next()`. An iterator for a list of length `n` has `n+1` possible cursor positions, as illustrated by the carets (`^`) below:

```
                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
 cursor positions:  ^            ^            ^            ^                  ^
```

Note that the `remove()` and `set(Object)` methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to `next()` or `previous()`.

Here are some examples of how to use a `ListIterator`. Given `myList` is the `ArrayList<String>  ["G", "C", "B", "K", "E", "A", "M"]`

Each of the following sections of code is separate. In each case, we start with the original list.

```
ListIterator<String> scan = myList.listIterator ();
while (scan.hasNext ())
   System.out.print(scan.next());
```

would output: `GCBKEAM` since this iterator goes through the whole list

```
ListIterator<String> scan = myList.listIterator (3);
while (scan.hasNext ())
   System.out.print(scan.next());
```

would output: `KEAM` since this iterator starts at position 3

```
ListIterator<String> scan = myList.listIterator (myList.size());
while (scan.hasPrevious ())
   System.out.print(scan.previous());
```

would output: `MAEKBCG` since this iterator goes through the list backwards

In the next example, we can remove elements as we check them by using the remove method. This method will remove the last element returned by next().

```
ListIterator <String> scan = myList.listIterator ();
while (scan.hasNext ())
{
    String nextElement = scan.next ();
    if (nextElement.compareTo ("D") <= 0)
        scan.remove ();
}
System.out.print(myList);
```

would output: `[G, K, E, M]` since the loop removes all elements <= "D"

As you can see from the above examples, the ArrayList class includes methods to return an Iterator or ListIterator object for the given List. For the ListIterator you can also specify where you want the ListIterator to start. Remember this value is not the starting index but the cursor position in the List (see examples on the previous page). Here are some of the methods that we can use with Iterators and ListIterators.

Iterator Methods

Since ListIterator extends Iterator, the following methods are available for both Iterators and ListIterators.

`boolean hasNext()`
Returns true if the iteration has more elements.

`E next()`
Returns the next element in the iteration.

`void remove()`
Removes from the underlying collection the last element returned by the iterator.

ListIterator Methods

`void add(E element)`
Inserts the specified element into the list.

`boolean hasPrevious()`
Returns true if this list iterator has more elements when traversing the list in the reverse direction.

`E previous()`
Returns the previous element in the list.

`void set(E element)`
Replaces the last element returned by next or previous with the new element.

2.6 Questions, Exercises and Problems

1) Assume you have created a `StringBuilder` object using the following:

```
StringBuilder text = new StringBuilder ("ABCDEFG");
```

What will be the value in the `StringBuilder` called `text` after each of the following commands? Each section is separate (i.e. you can assume you are starting with the original value of `text` each time).

	`StringBuilder` command(s)	New value of text
a)	`text.delete (3, 5);`	
b)	`text.insert (4, "XYZ");`	
c)	`text.replace (2, 7, "XYZ");`	
d)	`text.setCharAt (1, 'X');` `text.deleteCharAt(2);` `text.setCharAt (4, 'Y');` `text.reverse ();`	

2) Predict the output of the following section of code:

```
StringBuilder strBuild = new StringBuilder (10);
System.out.println (strBuild.length() + " " +
                    strBuild.capacity());
strBuild.append ("Do you");
System.out.println (strBuild.length() + " " +
                    strBuild.capacity());
strBuild.append (" get this?");
System.out.println (strBuild.length() + " " +
                    strBuild.capacity());
```

3) Write a Java method called `searchAndReplace()` that searches a string (`str`) for another string (`search`) and then replaces all occurrences of `search` with a third string (`replace`). Your method should return the new `String` with all of the replacements made. Your code should be both simple and efficient. To test your method, you should also write a small main method. Use the following method heading:

```
private static String searchAndReplace(String str,
                          String search, String replace)
```

For example, the following code should set newS`tr` to:
`"He wore a black shirt and black pants."`

```
String sentence = "He wore a blue shirt and blue pants.";
String newStr = searchAndReplace(sentence, "blue", "black");
```

4) Java has another type of object called a `StringBuffer` that is very similar to a `StringBuilder`. What is the main difference between a `StringBuffer` and a `StringBuilder`? When should you use each type of object?

5) Write a Java method called `wordToNumber` that converts a word into a number using the corresponding numbers for each letter on a standard phone keypad (given on the right). Your method should have a single String parameter (word to convert) and it should return the corresponding number as a String. For example, given "Hello", your method should return the number "43556", since H is 4, e is 3 etc. You can assume that the given word will contain only lower and upper case letters.
Note: Most numbers have 3 corresponding letters, but both 7 and 9 have 4 – so be very careful. Your code should be both simple and efficient. Here are some more examples:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
*	0	#

Given word parameter	Corresponding Number Sequence (Return value)
"Wonderful"	"966337385"
"Elephant"	"35374268"
"Corresponding"	"2677377663464"
"MNOPQRSTUV"	"6667777888"

6) Complete the code for a method called `encodeWord` that creates a coded word using a simple substitution code where 'A'=1, 'B'=2, 'C'=3, … 'Z'=26. Each word will only contain uppercase letters A to Z. The number codes in the each coded word should be separated by a '`-`'. For example `encodeWord("ELEPHANT")`should return: `5-12-5-16-8-1-14-20` Your code should be both simple and efficient. Include comments to explain your code.

```
/** Encodes a given word using a simple substitution code where
  * 'A'=1, 'B'=2, 'C'=3 ... 'Z'=26.
  * @param word the word to encode
  * Precondition: word will contain only uppercase letters
  *               and will have a least one letter.
  * @return the encoded word
  */
 public static String encodeWord (String word)
 {
     // Complete this code
 }
```

7) Write the code (including full comments) for a method called `noMultipleChars`. This method should remove extra characters from the given `String` parameter, returning a `String` that has no adjacent characters that are the same. It should remove only the extra characters. All of the remaining characters should be in their original order. For example: `noMultipleChars ("MISSISSIPPI")`would return `MISISIPI` and `noMultipleChars ("DDXXXXBBBXAA")`would return `DXBXA`. Your code should be very efficient. Here is the method heading to help get you started.

```
      public static String noMultipleChars (String str)
```

8) Predict the output of the following section of code:

```
ArrayList<String> animals = new ArrayList<String>(5);
animals.add("Elephant");
animals.add("Badger");
animals.add("Snake");
animals.add("Alligator");
animals.add("Giraffe");
animals.add(2, "Walrus");
System.out.println(animals);

System.out.println(animals.size());

Collections.sort(animals);
System.out.println(animals);

System.out.println(animals.get(2));

for (String nextAnimal : animals.subList(1, 4))
      System.out.printf("%s ", nextAnimal);
System.out.println();

System.out.println(animals.indexOf("snake"));

System.out.println(animals.lastIndexOf("Badger"));

System.out.println(animals.contains("Giraffe"));

animals.set(3, animals.remove(0));
System.out.println(animals);

System.out.println(animals.isEmpty());
animals.clear();
System.out.println(animals.isEmpty());
```

9) Assume you are given a `Student` class that keeps track of information about a student. You can also assume that this class includes a public method called `isGraduating()` that returns `true` if a student is graduating. Given `studentList` is a `ArrayList` of all of the `Student` objects in a school, write the section of Java code to remove all of the graduating students from this list. Be careful.

10) When you add or remove elements to/from an `ArrayList`, you may need to shift elements within the list. For each of the following sections of code, calculate the exact number of data shifts required within each `List`.

a) Assume that `listOne` contains 6 elements to start.

```
while(!listOne.isEmpty)
{
   listOne.remove(0);
}
```

b) Assume that `listOne` starts with 5 elements and `listTwo` starts with 2 elements.

```
while(listOne.size() > 1)
{
   listTwo.add(0, listOne.remove(1));
}
```

11) Assume you have two large `ArrayLists` of `Strings`. Complete the code for the method given below that finds and returns an `ArrayList` of all of the `Strings` that are in the first list but not in the second list. For example, given `first` is `["cat", "dog", "badger", "deer"]` and second is `["dog", "blue jay", "deer", "snake"]`, then `firstOnly(first, second)` should return the `ArrayList` `["cat", "badger"]`. Don't forget to include comments to explain your code. Here are the introductory comments and method heading, to help get you started:

```
/** Finds and returns an ArrayList of Strings that are
  * in the first list but not in the second list.
  * @param first the first ArrayList of Strings
  * @param second the second ArrayList of Strings
  * Postcondition: the contents of each list should remain unchanged
  * @return an ArrayList of all of the elements in the first that
  *        are not in second
  */
static ArrayList <String> firstOnly (ArrayList <String> first,
                                     ArrayList <String> second)
```

Chapter 3 – Objects and Searching Lists of Objects

In Java, in addition to using built in classes such as `Strings`, we can also create our own classes/objects. Just like real world objects, objects in Java have characteristics (data) and behaviour (methods). When we create a new Java class we create a template that defines an object's data and methods. Using the class code, we can create "instances" of a particular class and then manipulate these new objects.

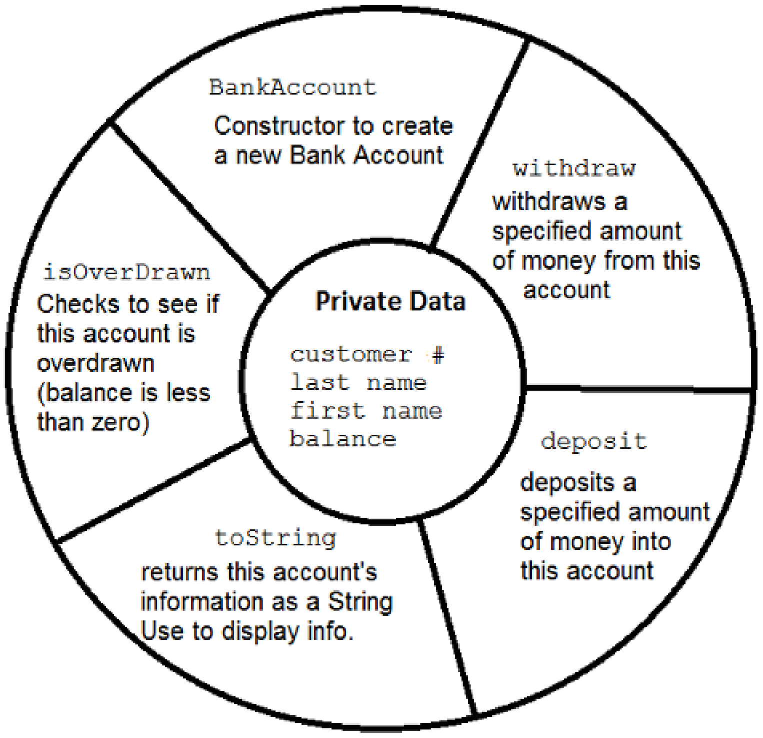
3.1 Encapsulation

Putting an object's data and methods together in a class is known as "encapsulation". Another important part of encapsulation is that an object should be responsible for its own data. Therefore, in most cases, we make an object's data private and then use public methods to access this private data. This is known as "data-hiding" since the details of how the data is stored in an object are hidden from the user of the object.

When using an object, the user focuses on the behaviour of the object or what they want to do with it. For example, when using a `String` object we can use all of the `String` methods such as `substring`, `charAt` and `length` without worrying about how the individual characters of the `String` are actually being stored. In fact, if the programmer who developed the `String` class code changed how the `String` data was stored, it would not affect how we use this class.

The diagram on the right shows the data and methods for a simple `BankAccount` class. The structure of the diagram highlights the encapsulation concept by showing the data and methods together within the same outer circle or "capsule".

It also shows how the public methods such as `withdraw` and `deposit` on the outer layer protect the private data such as `customer #` and `balance` on the inner layer. If the users of this `BankAccount` object want to look at or change the private data they need to go through the public methods. This helps each object remain responsible for its own data.


BankAccount
Constructor to create a new Bank Account
withdraw
withdraws a specified amount of money from this account
isOverDrawn
Checks to see if this account is overdrawn (balance is less than zero)
Private Data
customer #
last name
first name
balance
deposit
deposits a specified amount of money into this account
toString
returns this account's information as a String Use to display info.

3.2 Object Example: A BankAccount class

In the following example, we will be creating a simple BankAccount class. Start up Eclipse and create a new Java project. Call your new project BankAccountDemo.

Add a new class called BankAccount to your project. Do not include a main method in this new class. You should get the following code:

```
public class BankAccount
{

}
```

Our first step is to add the data fields to our new class. To keep it simple, we will add a name field and a balance field. Add this code to the BankAccount class.

```
private String name;
private double balance;
```

We will keep both of our data fields private so that they can't be directly accessed from outside the class without going through a public method. In simple examples this doesn't always seem necessary but it is an important principle in object oriented programming.

We should also get in the habit of including comments in our code. Each class should have a comment that identifies what the class is used for as well as the author and version. Add the following comments to the top of the BankAccount class:

```
/** Keeps track of a Bank account information including the
  * name (owner of the account) and the current balance
  * @author ICS4U
  * @version September 2014
  */
```

Our next step is to add some methods to our new class. One of the first methods we need for every class is a constructor that will be used to create a new object. A constructor usually defines the initial values of an object's data fields. Add the following code inside the BankAccount class. Once again, don't forget to include the comments.

```
/** Constructs a new BankAccount object with the
  * given name and balance
  * @param newName the name of the account holder
  * @param openBalance the opening balance for the new account
  */
 public BankAccount(String newName, double openBalance)
 {
    name = newName;
    balance = openBalance;
 }
```

You may have noticed that the constructor has the same name as the class. You will also notice that it is public and it has no return value (not even void). The parameters of the constructor are used to pass in the initial values for the data fields.

To test this code, we will create another class with a `main` method. Add a new class to your project called `BankAccountMain`. To make sure that the new class has a `main` method, check off the box to add the `main` method stub when creating the new class. Add the following code to the `main` method in the `BankAccountMain` class. Remember, there is no `main` method in the `BankAccount` class.

```
BankAccount firstAccount = new BankAccount ("Fred Flintstone", 345.67);
System.out.println(firstAccount);
```

Run your `BankAccountMain` program and you should get something like the following: (the number after the @ sign may be different).

```
    BankAccount@19821f
```

What about the name and the balance? When you display an object using `println`, the object's `toString` method is called to display the object as a `String`. Since we didn't define the `toString` method in our `BankAccount` class, the `toString` method for the `Object` class is called instead. By default, all objects extend the `Object` class and therefore we inherited `Object`'s `toString` method. The `toString` method in the `Object` class, returns a string consisting of the name of the class of which the object is an instance and the unsigned hexadecimal representation of the hash code of the object (separated by @). Since this doesn't tell you much about the individual object, you should always override the `toString` method when creating a new class.

By adding the following `toString` method to our `BankAccount` class we will "override" the `Object` class' version. The new `toString` method provides more information about the contents of each of our `BankAccount` objects. Here is the method with comments:

```
   /** Returns the BankAccount information as a String
     * @return the bank account name and current balance
     */
   public String toString()
   {
      return String.format("%-20s $%,10.2f", name, balance);
   }
```

In this method, the `String.format` command creates a new `String` using the same format codes used in `printf`. Before testing your code, create and display another `BankAccount` in the `main` program of the `BankAccountMain` class. An example is given on the next page.

```
BankAccount secondAccount = new BankAccount ("Betty Rubble", 1456.70);
System.out.println(secondAccount);
```

When you run your program, the output should be:

```
Fred Flintstone      $    345.67
Betty Rubble         $  1,456.70
```

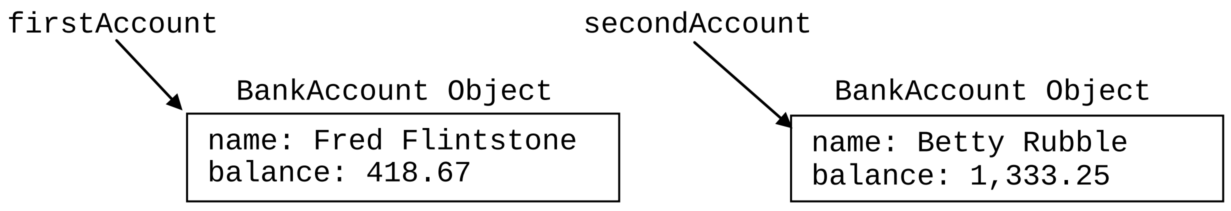
Let us add more behaviour (methods) to our `BankAccount` class. The `main` program code given below assumes two more `BankAccount` methods. Add this code to your `main` program and then update the `BankAccount` class so that it can handle this new code. You will need to add 2 new methods (`deposit` and `withdraw`). Look over the sample code to figure out the required parameters and code for these new methods. Don't forget to add full javadoc comments to both of the methods. Here is the code to add to your main program:

```
     // Update the balances
     firstAccount.deposit(73.00);
     secondAccount.withdraw(123.45);
     System.out.println(firstAccount);
     System.out.println(secondAccount);
```

After you add the `deposit` and `withdraw` methods to your `BankAccount` class, the above code should give you the following additional output:

```
Fred Flintstone      $    418.67
Betty Rubble         $  1,333.25
```

In the above examples, we created two instances of the `BankAccount` class (two `BankAccount` objects). Each object has its own name and balance. A memory representation of our two objects is shown below. `firstAccount` and `secondAccount` are `BankAccount` references that refer to the two `BankAccount` objects shown. (Note: In this diagram, to keep things simple, the `String` variable `name` is shown like a primitive variable).



In addition to the 2 instance variables (variables that are created for each individual instance of a class), we can also create class or `static` variables. Class variables are shared by all instances of the class. Assuming that the interest rate is the same for all `BankAccounts`, we can add a static (class) variable to the `BankAccount` class.

Add in the following code above the declarations for the name and balance fields at the top of the BankAccount class. This code declares and initializes the static variable interestRate. It is ok to initialize static variables when they are being declared but you should always initialize instance variables in the constructor.

```
private static double interestRate = 0;
```

To allow this static variable to be changed from outside of the class, we need to add the following public static method to the other BankAccount methods.

```
/** Sets the current interest rate for all bank accounts
  * @param newRate the new rate to set
  */
public static void setInterestRate(double newRate)
{
   interestRate = newRate;
}
```

Finally, add the following code to the main program.

```
BankAccount.setInterestRate(2.4);
firstAccount.addInterest();
System.out.println(firstAccount);
```

Notice that, since setInterestRate is a static (class) method that is setting the interest rate for all of the BankAccount objects, we call this method using the class name (BankAccount.setInterestRate) instead of using an instance reference such as firstAccount. To get this code to work you will need to add an instance method called addInterest to the BankAccount class. This new method should calculate the interest and then add it to the balance. Assume the interest rate given is a yearly percentage rate, but that interest is calculated monthly on the current balance. You are allowed to use the static variable interestRate in the instance method addInterest. Do you think you can use an instance variable inside a static method?

Add another method to the BankAccount class called isBalanceOver that checks to see if the current balance is over a certain amount. Here is the method heading to help get you started. Add in the body and full comments for this method.

```
public boolean isBalanceOver(double threshold)
```

Test your method using the following main method code:

```
if (firstAccount.isBalanceOver(500))
    System.out.println("Your bank fees will be waived");
else
    System.out.println("Your bank fee is $3.00");
```

Try changing firstAccount to secondAccount and see what happens.

3.3 `ArrayLists` of User Defined Objects

Since we will usually want to deal with more than one `BankAccount` object at a time, we can create an `ArrayList` of `BankAccounts`.

Continuing with the Bank Account project created in the previous section, add the following code to declare and create an `ArrayList` of `BankAccount` objects to the main method of your `BankAccountMain` class.

```
ArrayList<BankAccount> accountList =
                                new ArrayList<BankAccount>();
```

Next, we want to create some new `BankAccounts` and add them to `accountList`. Add the following code to your `BankAccountMain` class. Notice that we can create and add an account all in one step.

```
  accountList.add(new BankAccount("Ace Ventura", 632.25));
  accountList.add(new BankAccount("Liz Arden", 8885.78));
  accountList.add(new BankAccount("Tim Horton", 1345.67));
  accountList.add(new BankAccount("Clara Hughes", 895.24));
  accountList.add(new BankAccount("Jack Bauer", 217.12));
  accountList.add(new BankAccount("Lara Croft", 4317.39));
```

Next, we can use the “for each” loop to display all of the accounts in `accountList`.

```
     for (BankAccount nextAccount: accountList)
        System.out.println(nextAccount);
```

This should give you the following output:

```
Ace Ventura          $    632.25
Liz Arden            $  8,885.78
Tim Horton           $  1,345.67
Clara Hughes         $    895.24
Jack Bauer           $    217.12
Lara Croft           $  4,317.39
```

As a special promotion, the bank has decided to add $25 to the balance of every customer who has more than $1000 in their account. Write the Java code to apply this special promotion to all of the `BankAccounts` in `accountList`. Hint: Use the `isBalanceOver` method in your `BankAccount` object. Your new output should be:

```
Ace Ventura          $    632.25
Liz Arden            $  8,910.78
Tim Horton           $  1,370.67
Clara Hughes         $    895.24
Jack Bauer           $    217.12
Lara Croft           $  4,342.39
```

3.4 Searching an `ArrayList` using a Linear Search

Now that we have an `ArrayList` of `BankAccounts` we might want to search this `List` to find a certain customer. Continuing with the code from the previous section, add the following code to the `BankAccountMain` class. This code uses the `ArrayList` method `indexOf` to search an `ArrayList`. Before running this code, try to predict the output.

```
BankAccount target = new BankAccount("Tim Horton", 1345.67);
int index = accountList.indexOf (target);
System.out.printf("Index of %s is: %d%n", target, index);
```

Were you surprised with the result? Before reading ahead, can you explain what happened?

If you remember from Chapter 2, the `indexOf` method uses the `equals` method to check for a match when searching a `List`. Since we didn't define an `equals` method in our `BankAccount` class, the inherited `equals` method was used which only returns true if the object references are equal, not the contents of the object. To fix this problem, add the following method to your `BankAccount` class. Don't forget to include the full comments.

```
/** Checks to see if the given Object is a BankAccount
  * with the same name field as this BankAccount
  * @param other the object to compare to this BankAccount
  * @return true if the given object is a BankAccount with
  *         the same name as this BankAccount, false otherwise
  */
public boolean equals (Object other)
{
  // Uses the "instanceof" operator to check if other
  // is a reference to a BankAccount object
  if (!(other instanceof BankAccount))
     return false;

  // We now know that other must refer to a BankAccount
  // object so we can cast it to a BankAccount reference.
  // This step is needed in order to be able to access the
  // name field since other.name would give an error
  BankAccount otherAccount = (BankAccount) other;
  return this.name.equals(otherAccount.name);
}
```

You many have noticed that this method has an `Object` parameter. Since we can compare non `BankAccount` objects to `BankAccounts` we need to include an `Object` parameter. If the object that we are comparing to is not an instance of `BankAccount` we will return `false`. Once we know the object is a `BankAccount`, we can cast it and then access the name field. You may have also noticed the `this` modifier used on the name field (i.e. `this.name`). The `this` modifier can be used to more clearly specify "this" object's instance variable.

Running the original code after adding the `equals` method to the `BankAccount` class, you should get the following result:

```
Index of Tim Horton           $  1,345.67 is: 2
```

The `indexOf` method used above performs a simple “linear” search on our ArrayList. It goes through each element of the list looking for a match using the object “`equals`” method. If no match is found (which was the case before adding in `equals`), the method returns `-1`.

For small lists the linear search is fairly fast. However, for larger lists, this search would be very slow. For example, if we were searching a list of 10,000 elements we would need on average 5,000 comparisons in order to find the matching element. In the worst case (which would happen every time the element was not found), we would need 10,000 comparisons. In Big O notation, the linear search is O(n).

In the next section we will look at a faster search called a binary search.

3.5 Searching an `ArrayList` using a Binary Search

As we saw in the previous section, a linear search can be very slow for larger lists. For example, if you were looking for a name in a phone book, a linear search would start at the beginning and then look through every name in the book one by one until a match was found.

A faster way to look for something in a list is to use a binary search. In a binary search we assume that we start with a sorted list. At the first step we compare our target to the middle element of the list. If we don’t get a match we check to see if the target is less than or greater than the middle. If the target is less than the middle, we can eliminate the upper half of the list and if the target is greater than the middle, we can eliminate the lower half of the list. We then look at the middle of the remaining half and continue this process until we find a match.

At each step we are eliminating roughly half of the elements of the list. So, for a list of 10,000 elements, we would reduce the number of elements to 5,000 after the first step, 2,500 after the second step, 1,250 after the third step etc. In total we would only need around 14 steps ($\log_2 10{,}000$) to search the list. This is a lot less than the 5,000 average comparisons needed in a linear search. Each binary search comparison is a little more complicated since we need to find the position of the middle and we have 3 possible outcomes (>, < or =), however it is still a lot faster than a linear search especially for larger lists. Because we are halving the size of the list to search at each step, the run time of binary search in Big O notation is O(log n).

Another important point to remember is that a binary search only works if the list is in order. So, for a one time look up on an un-ordered list, a linear search would be faster since we wouldn’t need the extra time to sort the list. However, if we need to perform multiple searches on a list, adding a step to sort the list initially is worth it since we will be able to use the binary search. Also, we will see later that if we add new items to a list in the correct order, sorting is not always necessary.

Here is the code to use the built in binary search. The methods to sort the list (remember a binary search will only work with a sorted list) and for the binary search are both static methods in the `Collections` class. This code should replace the linear search code in the `BankAccountMain` class.

```
Collections.sort(accountList);
BankAccount target = new BankAccount("Tim Horton", 1345.67);
int index = Collections.binarySearch(accountList, target);
System.out.printf("Index of %s is: %d%n", target, index);
```

To make the above code work, we still need to add in some extra code to the `BankAccount` class. In order to sort and binary search our `List` of `BankAccount` objects using the `Collections.sort` and `Collections.binarySearch` methods, we need to make the `BankAccount` class `Comparable`. All `Comparable` objects are required to include a `compareTo` method that can be used to determine the order of two objects. To make the `BankAccount` class `Comparable`, update the class heading to the following:

```
public class BankAccount implements Comparable<BankAccount>
```

Since the `BankAccount` class is now `Comparable` we need to include a `compareTo` method that tells the computer how to compare two `BankAccount` objects. Add the following `compareTo` method to your `BankAccount` class:

```
/** Compares this BankAccount to another BankAccount
  * by comparing the name fields in each BankAccount.
  * @param other the BankAccount to compare to this BankAccount
  * @return a value < 0 if the name in this account comes
  *         alphabetically before the name in the other account,
  *         a value > 0, if this name comes after the other name
  *         and 0, if the names in the two accounts are the same
  */
public int compareTo (BankAccount other)
{
    return this.name.compareTo(other.name);
}
```

Just like the `compareTo` in the String class, the `compareTo` method should return a value < 0 if this object is less than the object you are comparing it to, a value > 0 if it is greater than and a value of zero if the two objects are equal.

Now go back to your main program and check if the sort and binary search work correctly. Since we used a single String (to keep things simple) for the customer's name, the list should be sorted by first names. The `compareTo` method in a `Comparable` object defines the objects "natural" order. An object's natural order defines the typical order we would put the object in.

Now, try changing the target to a name that is not in the list and see what happens. For example, change the target to the following:

```
BankAccount target = new BankAccount("Kim Weston", 1345.67);
```

Like `indexOf`, `Collections.binarySearch` returns a negative value when the target is not found, but instead of returning -1 each time, it returns a negative number which tells you where this element should be inserted in the list. Since you need a sorted list for the binary search, this can be very helpful. The actual return value turns out to be:

```
-insertionIndex – 1
```

The extra `–1` is needed to deal with the case when the insert index is 0 since -0 is still 0. For example, to keep the list in order, `"Kim Weston"` should be inserted at index 3, so the return value should be `-3-1= -4`. We can then do a similar operation to find the point of insertion. So to insert the missing target in the list we could use:

```
accountList.add(-index-1, target);
```

which would insert the target into the list in the correct position since `–(-4)-1` is `3`.

Here is an example of a binary search on a list of Strings. At each stage we need to keep track of the low index and the high index of the current section of the list that we are searching. When finding the middle index we look at the average value between the low index and high index (using integer division). If the target is greater than the middle, then the target must be in the upper half so we change low index to middle + 1. If the target is less than the middle, then the target must be in the lower half so we change high to middle - 1. Here is the list we are searching:

value	"A"	"C"	"E"	"F"	"G"	"H"	"J"	"K"	"M"	"N"	"O"	"P"	"Q"	"T"	"V"	"W"	"X"
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Searching for "G"

low index	high index	middle index	middle value	Notes
0	16	(0+16)/2 = 8	"M"	"G" < "M" so high = middle -1
0	7	(0+7)/2 = 3	"F"	"G" >"F" so low = middle +1
4	7	(4+7)/2 = 5	"H"	"G" < "H" so high = middle -1
4	4	4	"G" - found	Return 4

Searching for "R"

low index	high index	middle index	middle value	Notes
0	16	(0+16)/2 = 8	"M"	"R" > "M" so low = middle+1
9	16	(9+16)/2 = 12	"Q"	"R" >"Q" so low = middle +1
13	16	(13+16)/2 = 14	"V"	"R" < "V" so high = middle -1
13	13	(13+13)/2 = 13	"T"	"R" < "T" so high = middle -1
13	12	indices crossed	not found	Return -13-1 = -14

Notes: When low and high are equal we can still continue the search.
When the indices (low and high) cross (low > high), the element is not in the list.
The correct position to insert an element not found in the list is the <u>low index</u>.
Remember the binary search method returns: `-insertionIndex -1`

3.6 Comparing elements using `Comparators`

We saw in the last section that the `compareTo` method in a `Comparable` object defines the object's "natural" order. An object's natural order defines the typical order we would put the object in. However, in some cases, we may want to sort a list of objects using different criterion. We can do this by creating a new `Comparator`.

Add the following code to the bottom of your `BankAccount` class. This code will define an inner static `Comparator` class that will compare two `BankAccount` objects using their balances.

```
/** An inner Comparator class that compares two BankAccounts
  * by their balances
  */
private static class BalanceOrder implements Comparator<BankAccount>
{
    /** Compares the balances of two BankAccount objects
      * @param first the first BankAccount to compare
      * @param second the second BankAccount to compare
      * @return a value < 0 if the first BankAccount has a lower balance,
      *         a value > 0 if first BankAccount has a higher balance and
      *         0 if the balances of the BankAccount are the same
      */
    public int compare (BankAccount first, BankAccount second)
    {
        if (first.balance < second.balance)
            return -1;
        else if (first.balance > second.balance)
            return 1;
        else
            return 0;
    }
}
```

Note the `Comparator` uses a `compare` method with 2 parameters instead of the single parameter used by the `compareTo` method. Next we need to create a public static final instance of this new inner class that will be used by the `Collections.sort` or `Collections.binarySearch` methods. Add the following code to the top of your `BankAccount` class.

```
// Constant Comparator object for comparing BankAccounts by balances
public static final Comparator <BankAccount>
                  BALANCE_ORDER = new BalanceOrder();
```

Finally, to sort by this new criterion, update the sort method in the main program to:

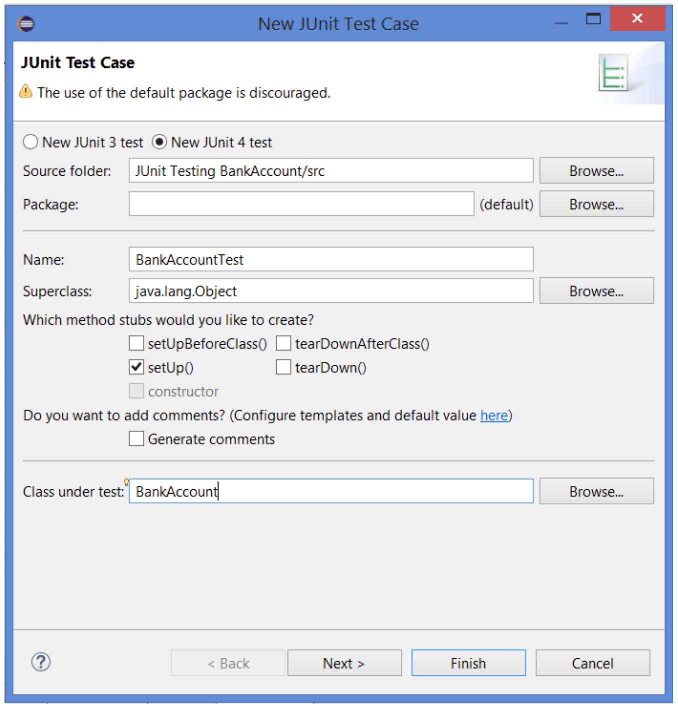
```
     Collections.sort(accountList, BankAccount.BALANCE_ORDER);
```

This code should sort the `List` of `BankAccounts` in ascending order by balance. How could you sort the accounts in descending order (highest balance first)?

3.7 Unit Testing with JUnit

We saw in chapter 1 that unit testing is used to test individual sections of program code. For example, unit testing would be used to test the code in our `BankAccount` class. In Eclipse we can create a `JUnit` class to test sections of another class' code. Below we will create a JUnit test to test the `withdraw`, `addInterest` and `isBalanceOver` methods in our `BankAccount` class. This code will also test the constructor and the `toString` method.

In your Bank Account project, right click on the `BankAccount` class and select `New->JUnit Test Case`. You should get the first screen on the left with default selections shown. Make sure that the `New JUnit 4 test` option is selected. Also, check that the default test program's name (e.g. `BankAccountTest`) is entered and the Class under test is set to `BankAccount`. Finally, you should make sure that the `setup()` method is selected under "Which method stubs would you like to create". When you select `Next`, you should get the screen on the right. On this screen, select the methods you want to test. Three methods are shown selected in the example.


New JUnit Test Case

JUnit Test Case

The use of the default package is discouraged.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: JUnit Testing BankAccount/src

Package: (default)

Name: BankAccountTest

Superclass: java.lang.Object

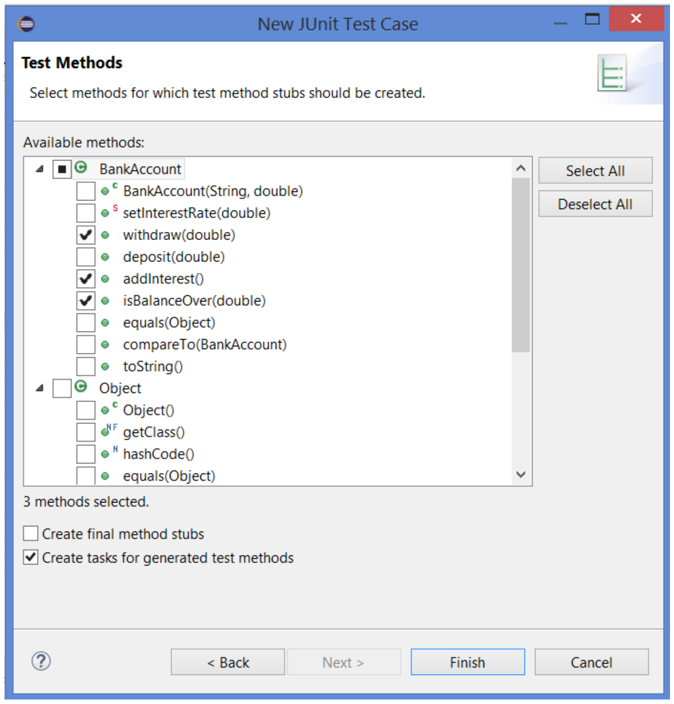
Which method stubs would you like to create?

- [ ] setUpBeforeClass()
- [ ] tearDownAfterClass()
- [x] setUp()
- [ ] tearDown()
- [ ] constructor

Do you want to add comments? (Configure templates and default value here)

- [ ] Generate comments

Class under test: BankAccount


New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- BankAccount
  - [ ] BankAccount(String, double)
  - [ ] setInterestRate(double)
  - [x] withdraw(double)
  - [ ] deposit(double)
  - [x] addInterest()
  - [x] isBalanceOver(double)
  - [ ] equals(Object)
  - [ ] compareTo(BankAccount)
  - [ ] toString()
- Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)

3 methods selected.

- [ ] Create final method stubs
- [x] Create tasks for generated test methods

When you select `Finish`, you should get starter code similar to the code below and on the next page. Fill in the missing code to complete each test. Be careful with the spacing for the expected `toString` output – make sure that it matches your spacing exactly. Here is the `BankAccountTest` code:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
```

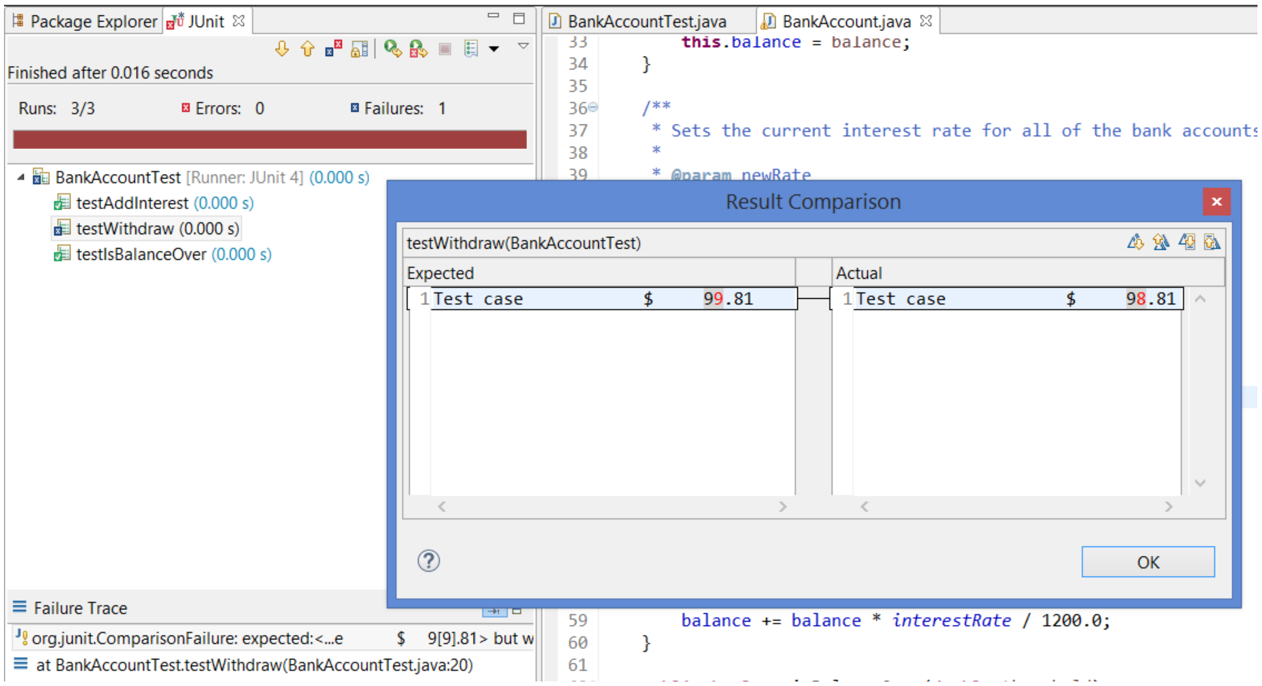
```
public class BankAccountTest
{
     BankAccount testAccount;

     @Before
     public void setUp() throws Exception
     {
          testAccount = new BankAccount("Test case", 123.45);
     }

     @Test
     public void testWithdraw()
     {
          testAccount.withdraw(23.64);
          assertEquals("Test case            $     99.81",
                         testAccount.toString());
     }

     @Test
     public void testAddInterest()
     {
          BankAccount.setInterestRate(5.0);
          testAccount.addInterest();
          assertEquals("Test case            $    123.96",
                         testAccount.toString());
     }

     @Test
     public void testIsBalanceOver()
     {
          assertTrue(testAccount.isBalanceOver(123.43));
          assertTrue(testAccount.isBalanceOver(123.44));
          assertFalse(testAccount.isBalanceOver(123.45));
          assertFalse(testAccount.isBalanceOver(123.46));
     }
}
```

In this code, the `@Before` before the `setup` method means this code is run once before each test. It is used to create a new `BankAccount` object. Then each test method (indicated with the `@Test` tag) tests one of the 3 methods that we selected to test. In each test method, the method is run using the `testAccount` object created in the `setup` method.

The `assertTrue` and `assertFalse` methods check to see if the given value is true or false. The `assertEquals` method has two parameters. The first parameter is the expected result and the second parameter is the actual result. It works for both objects and primitives. To handle `doubles` where precision may be an issue; you need to add a third `delta` parameter. The `delta` parameter is the maximum difference between the expected and actual for which both numbers are still considered equal. For example: `assertEquals(expected, actual, 0.001)`will be true if the difference between `expected` and `actual` is less than 0.001.

When you run your tests you get a report showing how many tests were run and the number of failures. It also shows the run time of each test case. You can check the reason for any failures by selecting the failed test case under the Failure Trace section. See sample below.

For this test, an error was added to the `withdraw` method on purpose.



Finished after 0.016 seconds

Runs: 3/3 Errors: 0 Failures: 1

- BankAccountTest [Runner: JUnit 4] (0.000 s)
  - testAddInterest (0.000 s)
  - testWithdraw (0.000 s)
  - testIsBalanceOver (0.000 s)

```
        this.balance = balance;
    }

    /**
     * Sets the current interest rate for all of the bank accounts
     *
     * @param newRate
```

testWithdraw(BankAccountTest)

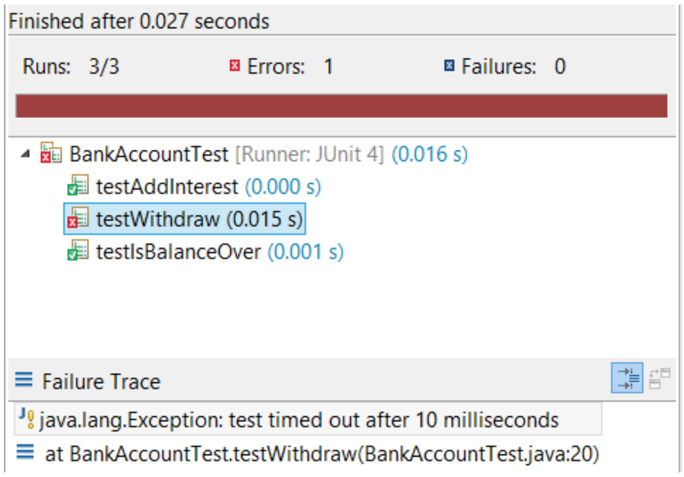
Expected	Actual
1 Test case $ 99.81	1 Test case $ 98.81

```
        balance += balance * interestRate / 1200.0;
    }
```

org.junit.ComparisonFailure: expected:<...e $ 9[9].81> but w

at BankAccountTest.testWithdraw(BankAccountTest.java:20)

The `@Test` tag also has some options to test for things such as thrown Exceptions and your method run time. For example: `@Test(timeout=n)` fails if the method takes longer than n milliseconds to complete. To test this feature a large loop was added to the `testWithdraw` method and the timeout was set to 10 milliseconds. In this case the timeout shows up as an error. See results on the right.



Finished after 0.027 seconds

Runs: 3/3 Errors: 1 Failures: 0

- BankAccountTest [Runner: JUnit 4] (0.016 s)
  - testAddInterest (0.000 s)
  - testWithdraw (0.015 s)
  - testIsBalanceOver (0.001 s)

java.lang.Exception: test timed out after 10 milliseconds

at BankAccountTest.testWithdraw(BankAccountTest.java:20)

For a list of more options and more information on JUnit testing you can check out the following on-line tutorial at:

http://www.vogella.com/tutorials/JUnit/article.html.

You may find some other JUnit tutorials on line as well.

3.8 Questions, Exercises and Problems

1) Explain each of the following Object Oriented Programming (OOP) terms:
    a) Encapsulation
    b) Data Hiding
    c) Instance of a Class

2) What three comments should we always include at the top of each class?

3) What is the purpose of an object's constructor method? What is the name and return type of a constructor?

4) In the main program, how do you call an object's constructor method?

5) What happens when we originally called `System.out.println(firstAccount)`? Give a complete explanation.

6) Why should you always override the `toString` method in each class that you create?

7) What is the difference between a `BankAccount` reference and a `BankAccount` object? Explain your answer.

8) What is the difference between an instance variable and a static (class) variable?

9) In the main program of the `BankAccountMain` program, how do you call a static (class) method and how do you call an instance method? Explain why there are different ways of calling each type of method.

10) We were allowed to use the static variable `interestRate` in the instance method `addInterest`. Can you use an instance variable inside a static method? Explain.

11) Why is `setInterestRate` a static (class) method and `addInterest` an instance method? Clearly explain your answer.

12) Instead of creating the `deposit`, `withdraw` and `isBalanceOver` methods, we could have created two methods called `getBalance` and `setBalance` to update and check each `BankAccount`'s `balance` directly. The main reason we don't usually create or use getter and setter methods in objects is that an important principal of OOP it is to make each object <u>responsible for its own data</u>. Why do you think it is important for an object to be responsible for its own data?

13) If we want to use `Collections.sort` to sort a `List` of objects, what type of object must `List` contain?

14) When we were searching a list using `indexOf` or `Collections.binarySearch`, would we get the same results if we changed the balance of the `target` object?

15) If an object implements `Comparable`, what method must this object include? Explain.

16) Explain how you would sort a `List` of objects in a different order than the object's natural order. Give a complete explanation.

17) In the code given above we sort the `List` of `BankAccounts` in ascending order by balance. How could you sort the accounts in descending order (highest balance first)?

18) Given the following partial class code for the Employee class:

```
public class Employee
{
    private static double minimumWage = 11.00;
    private String lastName;
    private String firstName;
    private int employNo;
    private double payRate;

    public Employee (String firstName, String lastName,
                     int employNo, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.employNo = employNo;
        this.payRate = payRate;
    }

    public Employee (String firstName, String lastName, int employNo)
    {
        // this() calls another constructor with the matching parameter
        this (firstName, lastName, employNo, minimumWage);
    }

    public static void setMinimumWage (double minimumWage)
    {
        Employee.minimumWage = minimumWage;
    }

    public String toString ()
    {
       return String.format("%s, %s; No: %d; Rate: $%.2f",
              lastName, firstName, employNo, payRate);
    }
}
```

Also, given the following main program code:

```
Employee second = new Employee("Donald", "Trump", 23456);
Employee.setMinimumWage(11.25);
Employee first  = new Employee("Homer", "Simpson", 12345, 18.50);
```

a) Draw a memory diagram (similar to the diagram in Section 3.2) to show the memory after running the main program code given above. In your diagram make sure you clearly label at least one example of the following:

i) An instance of the `Employee` class   ii) A reference to an `Employee` object
iii) an instance variable   iv) a class variable

b) Referring to the above code, give a specific example of:

i) A class method   ii) An instance method

Question 18 (continued)

c) Predict the output of the following lines of code in the main program:

i) `System.out.println(first);`
ii) `System.out.println(second);`
iii) `System.out.println(first.lastName);`

d) In the first constructor, we use "`this.`" in front of the name variable and later in the `setMinimumWage()` method we use "`Employee.`" in front of the `minimumWage` variable. Explain why these extra modifiers are required and why we use "`this.`" in one case and "`Employee.`" in the other case. Make sure you use correct terminology in your answer.

e) Assume you have an `ArrayList` of `Employee` objects called `employeeList`. Give the <u>specific Java code</u> you would need to add to the above `Employee` class if you wanted to sort this list using the command `Collections.sort(employeeList)`. Assume the "natural" order for an `Employee` object would be ascending alphabetical order by name (i.e. `lastName`, `firstName` alphabetically).

f) If you wanted to search the `employeeList` given in part e), using the `indexOf` method, what code would you need to include in the `Employee` class? Give the specific code.

19) For each of the following classes, suggest at least 3 data fields (characteristics) that you may want to include in each class:

a) a `Student` class
b) an `Address` class
c) a `Song` class
d) a `Food` class
e) a `Date` class (e.g. Oct 5, 2014)
f) a `Card` class (e.g. Jack of Spades)

20) For each of the following classes, think of all of the data fields (characteristics) and some of the methods (behaviour) that you would want to include in each class.

a) a `Deck` class (for playing cards)
b) a `Hand` class (also for playing cards)
c) a `ComplexNumber` class
d) a `Picture` class

21) A rational number (fraction) is a number with both a numerator and a denominator such as 1/3, 2/5 and 4 (a denominator of 1 is not shown). Design and write the code for a `RationalNumber` class. In addition to a few constructors and a `toString` method, you should include methods to add, subtract, multiply and divide `RationalNumbers`. You should also include code to compare two `RationalNumbers` to see which one is bigger. Hint: You should plan on always storing your `RationalNumbers` in lowest terms. You should also write a `main` program to test <u>all</u> of the methods of this new `RationalNumber` class.

22) On average how many comparisons are required to search a list of 15,000 items:

a) using a linear search?
b) using a binary search?

23) Clearly explain why the binary search is usually faster than the linear search.

24) Given the following list of String objects:

Contents	“A”	“C”	“E”	“G”	“I”	“K”	“L”	“M”	“O”	“Q”	“S”	“U”	“W”	“X”	“Y”
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Trace the `binarySearch()` method for each of the following cases by showing the value of lower, upper and middle after each step. You should also indicate the final return value.

a) Searching for “H”			b) Searching for “Q”		
lower	upper	middle	lower	upper	middle

Final Return Value: __________ Final Return Value: __________

25) Using the following code as an outline, write the code for a `binarySearch()` method that behaves like the `Collections.binarySearch()` method for a `List` of `String` objects. You should also write a small main program to test your code. You could also test this code in Question 28.

```
static int binarySearch (_________________ list, _____________ target)
{
    int lower = 0;
    int upper = _________________________________;
    while (________________________________)
    {
        int middle = _____________________________________________;
        int compareValue = target.compareTo (list.get(middle));
        if (compareValue > 0)
            ____________________________________;
        else if (compareValue < 0)
            ____________________________________;
        else
            ____________________________________;
    }
    return __________________________;
}
```

Once you get your method working for a `List` of `String` objects, you can modify your method heading so that it will work for a `List` of any `Comparable` objects. In this new generic method heading we use `E` to represent a general element type of the `List`. The new heading would be:

```
static <E extends Comparable<E>> int binarySearch (List<E> list, E target)
```

26) Complete the code for your own version of the `indexOf` method that searches a `List` of `String` objects. Your method should behave the same as the `indexOf` method in the `ArrayList` class. Since we can't add an instance method to the `ArrayList` class, your method should be a static method with the following heading:

```
static int indexOf (List<String> list, String target)
```

You can test your new `indexOf` method using the same main program that you used to test your `binarySearch` method or your code in Question 28.

27) Referring back to the `firstOnly` method in question 11 in Chapter 2, see if you can make your code faster by using a binary search.

28) Write a Java program that finds the frequency (number of occurrences) of all of the words in a text file. Your final output should be a list, sorted by number of occurrences (most popular word first), of the top 20 words (or less if there are less than 20 different words) in the file and their frequency (number of times they occur in the file). It should also include the total number of words in the file, the total number of different words in the file and the total run time to complete the task. Your final output should be nicely formatted and include appropriate headings. Example output is shown below:

```
Most Frequent Words -- File: sample.txt
Total number of words: 15342
Number of different words: 1432
Total Time: 56 milliseconds

        Word       Frequency
       ------      ---------
 1)      the          1001
 2)      and           654
 .        .             .
 .        .             .
```

Here are some hints/comments to help get you started:

a) In order to keep things simple, you can assume that a "word" is a consecutive group of alphabetic characters. For example: "don't" would count as 2 words ("don" and "t"). An easy way to extract individual words from the file is to use a Scanner with the delimiters (characters used to separate each word) set to all non-alphabetic characters. To set the delimiters, we use the command:

```
inFile.useDelimiter("[^\\p{Alpha}]+");
```

In this case `\\p{Alpha}` is an alphabetic character, `^` is to indicate "not" and the + means you want to match multiple instances of the delimiter (e.g. 2 spaces in a row) to avoid getting empty tokens.

Question 28 (continued)

b) When finding the frequency of each word, you should ignore case so "The" and "the" should both count as the same word. **Hint:** As you read in each word from the file, convert it to lower case. You can do this in one line:

```
String nextWord = inFile.next().toLowerCase();
```

c) You should create a `Word` object to keep track of each word and its frequency. You should plan out this class carefully to make sure that you have included all of the required data fields and methods. Don't include any getters and setters in this class. Also, make sure you include an `equals` method (with an `Object` parameter) – this method is needed if you are going to use `indexOf` to search your list. To use a binary search you will need to make your `Word` object `Comparable`. You should try this program using both the `indexOf` method and the `Collections.binarySearch` method. When using the binary search the current list of `Word` objects needs to be in order. To keep this list in order you do not need to the sort the list. Instead you have to make sure that you add any new `Words` not found in the list in their correct position. This can be done easily by using the return value from `Collections.binarySearch`.

d) Since we want the "natural" order of our `Word` objects to be alphabetical by the word (for our binary search and to match the `equals` method), you should add a `Comparator` to the `Word` class in order to sort the `Word` objects by frequency. This code will be similar to the `Comparator` we added to the `BankAccount` class.

e) You should test your program by creating a small one-page text file. Once you are confident that your program is working, you can use it to find the most frequent words in text files of public domain books such as "Alice in Wonderland" and "Moby Dick". These files will be available in the class folder.

29) Given a list of movies with the production start and end date for each movie, find the maximum number of movies that you can star in without a scheduling conflict. Assume that you cannot star in two movies with overlapping production dates.

The movie data will be in a file. The format of each line in the movie file:

Movie Name, production start date(D/M/Y), production end date(D/M/Y)

For example: `Movie #1, 17/9/2014, 3/1/2015`

We will be going over the algorithm to solve this problem in class. To solve this problem you should create two new classes – a `Movie` class and a `Date` class. Before starting on the code, you should make a list of all of the required data fields (characteristics) as well as the required methods (behaviour) for each class. You should present this information to the teacher before typing up the code. When you write the code, don't forget to include proper Javadoc comments for each class and all of their methods (including the constructors). You should also include comments to explain the code in your main program and some of the larger methods in both the `Movie` and `Date` classes. You should also create two JUnit tests to test the code in these new classes.

Chapter 4 – Recursion, Sorting and Dynamic Programming

4.1 Recursion

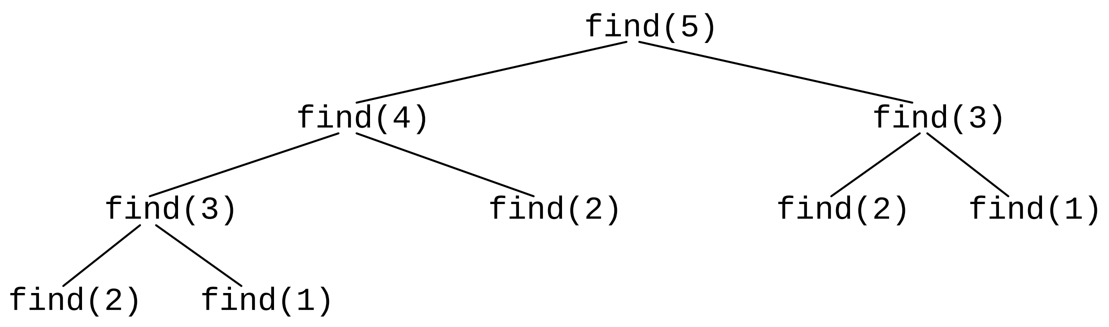
Recursion can be a difficult concept to understand but it is a powerful problem solving technique. If you don't understand the concepts presented after studying this material, make sure that you ask questions.

A recursive method is a method that calls itself. To help explain this process, let us look at a simple example of a recursive method.

```
public static int find(int number)
{
   if (number <= 2)
        return 1;
   return find(number - 1) + find(number - 2);
}
```

As you can see the code inside the `find` method calls the `find` method. So, if we call `find(3)`, we will end up calling `find(2)` and `find(1)`. Since `number <= 2`, both `find(2)` and `find(1)`will return 1. Then `find(3)`will add together these return values and return 2.

In this simple example, the recursion isn't too hard to follow. However, it can get more complicated. Try finding the value of `find(5)`. To keep track of the recursion, we can use a simple tree diagram to show the calling sequence.



Fill in the return value for each method call on the branches of this tree. If done correctly, you should get a final return value of 5 for `find(5)`. Try finding the value of `find(7)` on your own. Since you already know the value of `find(5)`, you don't need to complete the entire tree. You may have noticed that this method is not very efficient since we are recalculating many values. In fact, in the above tree, `find(2)`appears 3 times and both `find(3)` and `find(1)` appear 2 times. Later in the chapter we will look at a way of making this method more efficient by storing values so they don't need to be recalculated.

You may have noticed that this method finds the numbers in the Fibonacci sequence. Like all recursive algorithms we could have also written this code using a loop instead of recursion. In section 4.5 we look at a faster non-recursive way of finding the numbers in the Fibonacci sequence.

Like every recursive method, this method has two distinct parts.

1) **A base case or termination condition that causes the method to finish.** In the above example, the section of code that deals with the case when `number <= 2` is the base case. Note: A recursive method can have more than one base case.
2) **A case that moves the algorithm towards the base case and termination.** In the above example, when we call the method recursively the value of the parameter is reduced by 1 or 2 each time moving us closer to the base case.

Generally, recursion can be used when you can describe a process as a simpler case of itself. A good example of this is the factorial function. Recall that the factorial of a number n (given as `n!`) is just the product of all of the numbers between 1 and n.

$$n! = n\times(n-1)\times(n-2)...\times2\times1$$

To solve this problem using recursion, we note that `n!` can be expressed in terms of `(n-1)!`

$$n! = n\times(n-1)!$$

Since both `0!` and `1!` are `1`, our base case will return 1 if the parameter is 0 or 1. Here is the code for a recursive version of the factorial method. Since factorials can be very large the return type is `long`.

```
public static long factorial(int number)
{
     if (number <= 1)
        return 1;
     return number * factorial(number - 1);
}
```

As was mentioned earlier, all recursive methods can be written iteratively (with a loop). If you recall, the factorial method we wrote in Grade 11 used a loop. Although the factorial example given above helps to illustrate how recursion works, the non-recursive version would be the better choice since it is faster and it uses less memory. It is important to remember to use recursion wisely and only when it simplifies the code, like the example given in section 4.2.

One of the problems with recursion is the extra memory that it uses to keep track of all of the method calls. In Java, a new stack frame is created every time a method is called. This stack frame keeps track of a method's parameters and local variables as well as an operand stack that is used by the method. The stack frame also contains other data that is used by the Java Virtual Machine (JVM) to support operations such as returning from a method call.

When a method finishes running this frame is destroyed and the memory it used is freed up. However, if a method calls a new method before finishing (which happens during recursion), a new stack frame for the new method is created and it becomes the current frame. Even though only the frame of the current frame is active, the frames of methods that haven't finished are still taking up memory in the JVM stack. If we don't have a proper base case or if your recursion goes too deep, this stack memory that is filled with all of the stack frames of the unfinished methods will eventually get used up and we will get a `StackOverflowError`.

4.2 Recursion Example: Flood Fill

When you use the “paint can” in a paint program to fill in a bounded area of the picture, this is sometimes called a “flood fill” since the paint spreads out flooding the area being filled. This “flood fill” process can also be used to solve problems where we are looking for connected sections in a 2D array. Here is an example of a recursive implementation of the “flood fill” algorithm. Note: You need to be careful when filling very large regions recursively since there is a risk of a stack overflow error.

Assume that you are given the 2D character array on the right that represents a map of a section of ocean with a ‘`-`‘ representing open water and an ‘`X`’ representing land. Sections of land that are connected vertically or horizontally (but not diagonally) are part of the same island. Starting in a given row and column, your task is to find the size of an island by counting the number of `X`’s used to make up the island. For example, starting at row 4 and column 4, we can find an island of size 38 units. With all of the twists and turns in the island, the flood fill algorithm works quite well. Starting at any location on the island, we recursively keep looking in all 4 directions from each new starting point. To avoid looping back to count the same ‘`X`’ twice, we need to mark (fill with a lowercase ‘`x`’) each square as it is counted. We also need to count and return the number of marked squares in order to find the size of the island. This process of searching for all of the `X`’s on the grid will be a “depth first” search since we keep searching in one direction until we get a dead end.

```
-X-------------
-X---XXXXXXXXX-
-X-----------X-
-X---XXXXX-----
----XXX--XXXXX-
-XXXXXX-----XX-
-XX---------XX-
-XX---XXXX--XX-
-XX---XXXX----X
-XX---XXXX---XX
-XXXX--XX---XXX
----X--X---XXXX
```

Here is the heading and introductory comments for the recursive method that finds the size of an island starting in a given row and column. You can assume this method would be part of a `Map` class and that there would be an instance variable called `map` that would be a 2D array of characters representing the given map.

```
/** Checks if the square at the given point is part of an island.
  * If it is, mark this square and then check all surrounding squares
  * (up and down, left or right but not diagonally) to see if they
  * are part of this island
  * @param row the row to start checking
  * @param col the column to start checking
  * @return the number of squares (including this square) that are land
  *         squares connected to this square in a up or down or left or
  *         right direction and that have not already been counted
  * Postcondition: will mark any counted squares on the map with an ‘x’
  */
public int markIsland(int row, int col)
```

To complete the code for this method, our first step is to come up with a base case. Before turning to the next page, think about when we would need to stop the recursion.

You may have guessed that we want to stop the recursion when the given point to search is off the map or the square at the given point is not an 'X' (i.e. stop when this square contains open water or it has already been counted). Therefore our base case code is simply:

```
// Check if the given row and column is on the map and
// is part of an island, returning 0 when not valid
if (row < 0 || row >= map.length ||
    col < 0 || col >= map[0].length ||
    map[row][col] != 'X')
       return 0;
```

Notice that we need to check that the row and column are off the map first or we could get an out of bounds error when checking `map[row][col] != 'X'`

For our next step, we need to count and mark this square and then recursively search around this square for more sections of land connected to this island. We also need to add up all the return values of all of the searches around this square:

```
// Mark this square as counted ('x' is the fill colour)
map[row][col] = 'x';

// Return 1 for this square plus any squares
// of land around this square up or down
return 1 + markIsland(row + 1, col) +
           markIsland(row - 1, col) +
           markIsland(row, col + 1) +
           markIsland(row, col - 1);
```

Since we were only searching in the 4 spots around this given square, we just called each of the 4 possible searches directly. If we wanted to include diagonal searches for a total of 8 different spots we could use a loop and two direction arrays to reduce the amount of code. For example, at the top of the class we would include:

```
// 8 possible changes in row and column direction
final static int [] DROW = {-1, -1, 0, 1, 1,  1,  0, -1};
final static int [] DCOL = { 0,  1, 1, 1, 0, -1, -1, -1};
```

And then in the method we would replace the last section of code with:

```
// Set up the size and count this square
int size = 1;

// Add in the results from checking in all 8 directions
// around this square, returning the total size when done
for (int dir = 0; dir < DROW.length; dir++)
{
   size += markIsland(row + DROW[dir], col + DCOL[dir]);
}

return size;
```

4.3 Introduction to Sorting and Sort Stability

A common practice in many computer applications is to sort a list of elements into some kind of order. We can sort a list of elements into ascending order (lowest to highest) or descending order (highest to lowest). When sorting a list of objects, we can also sort based on different criteria. For example, we could sort a list of `Student` objects alphabetically by name or in descending order by their final average.

Since objects can have multiple fields, when sorting objects based on one field it is important not to change the order of like elements (elements that are equal based on the sorting field). For example, given a list of playing `Card` objects:

```
JC   7H    JS   QD   5D
```

If we wanted to sort this list by rank, we have two possible choices:

```
5D   7H   JC  JS    QD   or    5D   7H   JS  JC    QD
```

Both lists are correctly sorted by rank. However, in the sorted list on the left, the suit order of the two Jacks (`JC` and `JS`) is the same as the original list but in the list on the right the suit order of the Jacks has switched. In this case, during the process of putting the ranks in order, the suit order of two cards with the same rank was changed.

The sort on the left is a **stable sort** since the order of the like elements (Jacks in this case) is not changed. On the other hand, the sort on the right is a **non-stable sort** since it changed the order of like elements. When sorting a list of primitives that have only one data field, sort stability is not an issue since you can't distinguish between like elements. However, when sorting a list of objects that could have multiple data fields, using a stable sort is very important since you don't want to change the order of one field in the list when sorting based on another field. This is why Java uses the stable Merge Sort to sort lists of objects (see below).

In Java, we can use the built in sort methods `Arrays.sort` to sort arrays or `Collections.sort` to sort `Lists` including `ArrayLists`. When sorting objects, the "natural" order of an object is defined by the object's `compareTo` method. We can also pass a second `Comparator` parameter to both `Arrays.sort` and `Collections.sort` to change the criteria used to sort the list. Recall in the last chapter, we sorted a `List` of `BankAccount` objects in natural order by the customer's name and then we sorted the same list in order by the customer's bank balance using the `BankAccount.BALANCE_ORDER Comparator`.

These built in Java sorts use either the Quick Sort or the Merge Sort. Both of these algorithms are very fast `O(n log n)` sorts. In most cases, the Quick Sort is a quicker sort that uses less memory, however, unlike the Merge Sort, the Quick Sort is <u>not</u> a stable sort. Therefore, `Arrays.sort` (when sorting an array of objects) and `Collections.sort` (since `Lists` can only contain objects) use the fast and stable Merge Sort. On the other hand, when sorting an array of primitives, `Arrays.sort` uses the faster and less memory intensive Quick Sort since primitives only have one possible data field so sort stability is not an issue.

4.4 Sorting Algorithms

There are many different sorting algorithms available. We will be looking at the Selection Sort, Insertion Sort and Merge Sort. The algorithms presented will sort a list of n elements in ascending order (lowest to highest).

4.4.1 Selection Sort

The selection sort sorts a list by first selecting the smallest element in the list by scanning through the list and keeping track of the index of the smallest element. After checking the entire list, the smallest number is swapped with the first element in the list. Now that the first element is in order, the next step is to find the second smallest number in the list and then swap it with the second element in this list. This process continues until all of the elements are in order. **Note:** You don't need to find and swap the last element since if you sort the first n-1 elements the n^th^ element will be in order.

The following example shows the steps required to sort a list of integers using the selection sort. Here is the original list:

12	7	5	8	11	4	6

First we search through the list to find the smallest element (4 at index 5). Then we swap this element with the first element (12), giving the list shown below. It is important to swap only once on each pass after finding the next smallest number in the list.

4	7	5	8	11	**12**	6

With the first number (4) in place, we start searching from the 7, to find the next smallest number 5 (at index 2). Then we swap this element with the second element (7), giving the following list after the second pass.

4	**5**	**7**	8	11	12	6

Continuing with the third smallest etc. we get the following lists after each pass. Notice that after the `i`th pass, the first `i` elements in the list are guaranteed to be in order.

4	5	**6**	8	11	12	**7**	Swaps 6 and 7
4	5	6	**7**	11	12	**8**	Swaps 7 and 8
4	5	6	7	**8**	12	**11**	Swaps 8 and 11
4	5	6	7	8	**11**	**12**	Swaps 11 and 12

How many element comparisons did we need to complete this sort? How many swaps? Knowing the number of required comparisons and swaps can help us to determine a sort's run time (see Appendix C). These questions, including questions about the run time of this algorithm, will be left for exercises at the end of the chapter (see questions 12, 14, 16 and 23).

On the next page is the Java code for a Selection Sort method that sorts an `ArrayList` of `String` objects. Look over this code carefully to make sure that you understand how it works.

```
/** Sorts a List in ascending order (lowest to highest)
  * using the selection sort algorithm
  * @param list the List to sort
  */
public static void selectionSort(List<String> list)
{
   // Go through the list finding the next smallest element
   // at each pass placing it in its correct position
   for (int next = 0; next < list.size() - 1; next++)
   {
      // Find the position of the next smallest element
      int smallIndex = next;
      for (int check = next + 1; check < list.size(); check++)
      {
         if (list.get(check).compareTo(list.get(smallIndex)) < 0)
             smallIndex = check;
      }

      // Place the next smallest element in the next position
      // swapping it with the element currently in this position
      String temp = list.get(next);
      list.set(next, list.get(smallIndex));
      list.set(smallIndex, temp);
   }
}
```

4.4.2 Insertion Sort

In the insertion sort we build up larger and larger sorted sub lists on the left by inserting the next element on the right into the correct position in these sorted sub lists. The initial sorted list is just the first element. Then we keep this list sorted by inserting the next element into this list. For example, the first step is to insert the 2nd element into the list of 1 to create a sorted list of 2. Then the 3rd element is inserted into a list of 2 to create a sorted list of 3. This continues until all of the elements have been inserted in order. When inserting the next element in order, the algorithm works from right to left until it finds an element less than the element to insert or until it gets to the start of the list. As it is looking for the place to insert the next element, it shifts over the elements that need to be shifted.

The following example shows the steps required to sort a list of integers using the insertion sort. Here is the original list:

11	**7**	5	8	12	4	6

To start, the list of 1 on the left (11) is already in order. We now need to insert the 7 into this list. First we store the 7 in another memory location and then we compare 7 and 11. Since 11 is greater than 7, we shift the 11 over to the right. Then since we have reached the start of the list we insert the 7 into the correct first position getting the following list:

7	11	**5**	8	12	4	6

Next we need to insert 5 into the sorted list of 7 and 11. To insert 5, we shift both the 7 and the 11 and then insert the 5 in the first position getting the list on the next page:

5	7	11	**8**	12	4	6

Next we need to insert the 8 into the sorted list of 5, 7 and 11. In this case we only need to shift over the 11 and since the 8 is greater than 7, we can insert the 8 where the 11 used to be to get the following:

5	7	8	11	**12**	4	6

At the next step we insert 12 into the list of 5, 7, 8 and 11. Comparing the 12 and 11 we see the 12 is greater than 11 so it stays in the same spot and no shifting is required.

5	7	8	11	12	**4**	6

Then we insert the 4 into the list of 5, 7, 8, 11 and 12. In this case, we end up shifting over all of the values before inserting the 4 to get:

4	5	7	8	11	12	**6**

Finally, when we insert 6, we need to shift over the 7, 8, 11 and 12 and then insert 6 to get the final sorted list:

4	5	6	7	8	11	12

Here is the Java code for the Insertion Sort method that sorts an `ArrayList` of `String` objects. Look over it carefully to make sure that you understand how it works.

```java
/** Sorts a List in ascending order (lowest to highest)
  * using the insertion sort algorithm
  * @param list the List to sort
  */
public static void insertionSort(List<String> list)
{
   // Starting with the second element in the list, insert each element
   // into the sorted list to its left
   for (int next = 1; next < list.size(); next++)
   {
      // Get the next element to insert and compare it to the element
      // to its left. Keep comparing until you find a smaller element
      // or you get to the start of the list. If the element is larger,
      // shift it one to the right to make room for the element to insert
      String insert = list.get(next);
      int index = next - 1;
      while (index >= 0 && list.get(index).compareTo(insert) > 0)
      {
         // Shift a larger element to the right
         list.set(index + 1, list.get(index));
         index --;
      }

      // Insert the next element into its proper position
      list.set(index + 1, insert);
   }
}
```

See questions 12, 15, 16, 24 and 25 at the end of the chapter for some questions about the Insertion Sort.

4.4.3 Merge Sort

The merge sort is a recursive sort. At each step it sorts a given section (sublist) of the list by completing the following steps:

1) If the sublist size is 1 or less, this section of the list is sorted (base case)
2) Otherwise, divide this section of the list into two halves.
3) Sort each half by calling the merge sort recursively.
4) Merge the two sorted halves into a sorted list.

During this sort the recursive process breaks the list into n sublists, each of length 1. Since they only contain 1 element, these sublists are all sorted. It then merges these sublists of 1 into sublists of 2 and then these sublists of 2 are merged into sublists of 4 etc. This process repeats until the whole list is sorted. At each stage, the sorting occurs during the merging process. For example, given the following list:

11	7	5	8	12	4	9	6

At the first stage, the list is split into the sublists 11, 7, 5, 8 and 12, 4, 9, 6. The left list is then split into 11, 7 and 5, 8. Again, the left list is then split into 11 and 7. Since these lists of size 1 are now sorted, they can be merged back into a list of 2. So we get:

7	**11**	5	8	12	4	9	6

With the two left most elements now merged into a sorted sublist, the sort will break down the 5, 8 sublist into 5 and 8. Since these are already in order, no merging is required. Now with two sorted sublist of 2 on the left, we can merge these lists into a sublist of 4. At the merge stage, the first elements of the sublists (7 and 5 in the example) are compared and the lower element is copied into the merged list first. To keep things simple, we merge the elements into a new list that is then copied back to the original list. After merging the two sublists of 2 on the left into a list of 4 we get:

5	**7**	**8**	**11**	12	4	9	6

Now we repeat the same process with the sublist on the right, merging the 12 and 4 into a sublist of 2

5	7	8	11	**4**	**12**	9	6

Then we merge the 9 and 6 into a sublist of 2.

5	7	8	11	4	12	**6**	**9**

Next we merge these sublists of 2 into a sublist of 4.

5	7	8	11	**4**	**6**	**9**	**12**

Now we have 2 sorted sublists of 4 that need to be merged into a list of 8. As was mentioned, during this merging process we compare the next element of each sublist and move the lower element into the merged list. This process continues until all of the elements of one of the sublists have been moved into the merged list. We then copy over the remaining elements of the other sublist. You should complete the last merge of the above sublists of 4 on your own in the space provided.

Note: In the above example the size of the list was a power of 2 which made it easy to cut the list in half at each stage. In reality this is not always the case but it does not cause any problems if the sizes of the merging sublists are different.

Since the merging process described above requires extra memory to hold the elements being merged (before they are copied back to the original list), the merge sort takes up more memory than the sorts presented earlier. We could merge the sorted sections in place but it would be more complicated and will slow down the process.

Here is the Java code (3 methods) for the Merge Sort that sorts an `ArrayList` of `String` objects. Look over these methods carefully to make sure that you understand how each of them work. Pay special attention to the various index parameters – some are inclusive and some are exclusive.

You should also note some of the special cases that help speed up the code such as not merging two sublists that are already in order and therefore don't need to be merged and not copying over the second half being merged if the first half runs out of elements first.

```java
/**
 * Sorts a List in ascending order (lowest to highest) using the merge
 * sort algorithm. This version is included to match the other sorts
 * since the recursive mergeSort given below needs extra parameters
 * @param list the List to sort
 */
public static void mergeSort(List<String> list)
{
   mergeSort(list, 0, list.size());
}

/**
 * Sorts a section of a List in ascending order (lowest to highest).
 * Sorts the section between start (inclusive) and end (exclusive).
 * This method will call itself as well as the merge method
 * @param list the List to sort
 * @param start the start position of the list to sort (inclusive)
 * @param end the end position of the list to sort (exclusive)
 */
private static void mergeSort(List<String> list, int start, int end)
{
   // A list of 0 or 1 is already sorted
   if (end - start <= 1)
      return;

   // Sort the left and right halves of the list and then
   // merge these two halves into one sorted list
   int middle = (end + start) / 2;
   mergeSort(list, start, middle);
   mergeSort(list, middle, end);
   merge(list, start, middle, end);
}
```

```
/**
 * Merges two sorted sections of a List into one.
 * The merged elements are eventually stored in the given List.
 * Merges the section from [start, middle) and from [middle, end).
 * First section is from start (inclusive) to middle (exclusive).
 * Second section is from middle (inclusive) to end (exclusive).
 * @param list the List to merge in
 * @param start the start position of the first section to merge (inclusive)
 * @param middle the middle of the list. This will also be:
 *        the end position of the first section to merge (exclusive)
 *        the start position of the second section to merge (inclusive)
 * @param end the end position of the second section to merge (exclusive)
 */
private static void merge(List<String> list, int start,
                           int middle, int end)
{
   // If the sections to merge are already in order, we are done
   if (list.get(middle).compareTo(list.get(middle - 1)) >= 0)
      return;

   // Create a new list to store the merged sections. This takes extra
   // memory but is a lot easier than trying to merge the lists in place
   ArrayList<String> newList = new ArrayList<String>(end - start);

   // Merge the two lists taking the smallest element from either list
   // Continue this loop while both lists have elements remaining
   // This code is similar to the merge code we wrote in Grade 11
   // It is a little different since we are using ArrayLists
   int firstIndex = start;
   int secondIndex = middle;
   while (firstIndex < middle && secondIndex < end)
   {
     if (list.get(firstIndex).compareTo(list.get(secondIndex)) <= 0)
          newList.add(list.get(firstIndex++));
     else
          newList.add(list.get(secondIndex++));
   }

   // If the first half still has elements left,
   // copy the remaining elements from the first half
   while (firstIndex < middle)
     newList.add(list.get(firstIndex++));

   // If the second half still has elements left, they will
   // be in order so no copying is needed

   // Copy the merged elements back into the original list.
   int copyIndex = start;
   for (int index = 0; index < newList.size(); index++)
     list.set(copyIndex++, newList.get(index));
}
```

See questions 12, 17 and 26 at the end of the chapter for some questions about the Merge Sort.

4.5 Dynamic Programming (Optional)

Dynamic programming (DP) is a problem solving technique in which a problem is solved by breaking it into smaller sub problems or stages. We then use the solutions to the smaller sub problems (earlier stages) to help find the solutions to the larger sub problems (later stages). In some ways it is like recursion except it is usually faster because you are storing the solutions to the sub problems so you don't have to re-calculate them over and over again. Also, using DP we can work from the bottom up to avoid the extra overhead from the recursive method calls.

In section 4.1 we presented a recursive solution to finding numbers in the Fibonacci sequence. We also noted that this solution was not very efficient because it recalculated the same results more than once. Since most Fibonacci numbers are defined in terms of other lower Fibonacci numbers, this is a good fit for DP. Below we will look at two different ways of finding a solution to this problem more quickly.

One way to avoid recalculating values in the Fibonacci problem is to store the results as they are calculated. Assuming we want to find the first 50 Fibonacci numbers we can create a static array of 51 elements (since arrays are zero based) to store these results. Since Fibonacci numbers can get quite large we will use an array of `long`.

```
static long [] fibonacci = new long[51];
```

Then we modify our recursive method to take advantage of this new array.

```
public static long fibonacci (int n)
{
   if (n <= 2)
      return 1;

   // If we haven't already found the nth fibonacci number,
   // find it and put the result into the array
   if (fibonacci[n] == 0)
   {
       fibonacci[n] = fibonacci (n - 1) + fibonacci (n - 2);
   }

   // Return the result (either just found or found earlier)
   return fibonacci[n];
}
```

Initially the default values in the `fibonacci` array are 0. We will fill in these values as needed. Since this code does not recalculate any results it is much faster than the code in 4.1, especially for larger values of n.

This idea of storing the results of our calculations is known as "memoization". This recursive solution is also a "top-down" dynamic programming solution. The original and more traditional dynamic programming technique is a "bottom up" approach.

For the “bottom up” DP approach we use the same array to store the first 50 Fibonacci numbers. After assigning the first two known Fibonacci numbers to the array, we use a loop to fill in the rest of the numbers based on the previous results. This is a DP approach since we are using the solution for the smaller Fibonacci numbers to find the solution for a larger Fibonacci number. Here is the code:

```
// Set up an array to keep track of the Fibonacci numbers
long[] fibonacci = new long[51];

// Set the values of the first 2 Fibonacci numbers
fibonacci[1] = 1;
fibonacci[2] = 1;

// Calculate the next 48 Fibonacci numbers based on the previous values
for (int n = 3; n <= 50; n++)
   fibonacci[n] = fibonacci[n - 1] + fibonacci[n - 2];
```

To find the value of a Fibonacci number we can look up the value in the array using `fibonacci[number]` instead of calling the method.

So which approach should we use “top down” or “bottom up”? It depends on the problem. The “bottom up” approach has the advantage that it can be faster since you don’t have the overhead from the recursive method calls. This is especially true if many of the sub problems are revisited. The advantage of the “top down” approach is that it computes the sub problems only when necessary, which may be faster in some cases.

Note: The following discussion of the knapsack problem is based on work presented by S Dasgupta et al. in their *Algorithms* book.

Another good example to show how and when to use dynamic programming is the knapsack problem. A knapsack problem is a classic problem in computer science. The basic idea is that you have a “knapsack” that has a limited capacity. You are also given a collection of items with each item having a given weight and value. Your task is to select the optimal combination of the items in order to maximize the value of the items in your “knapsack” without exceeding its capacity. For example, assume we are given the following 4 items and the capacity (W) of your “knapsack” is 11 kg.

Item	Weight (kg)	Value ($)
1	5	14
2	3	9
3	2	5
4	6	17

We can consider two types of knapsack problems. The first type allows repetition which means we can select more than 1 of each item. Given the above data, what is the optimal solution to this problem?

To solve this problem using DP, we define:

`Value(w)` = maximum value of the items in the knapsack given a capacity of w.

Given the current capacity is w, we can look at all of the items with weight less than w and then decide whether to include this item or not. If we include the item we will increase the value of the items in our knapsack but we will need to decrease the remaining capacity. This leads to the following relationship that defines the optimal solution to a larger problem in terms of the optimal solution to a smaller problem.

$\text{Value}(w) = \max\{\text{Value}(w-w_i) + v_i\}$ for all i such that $w_i <= w$

where w_i and v_i are the weight and the value of the i^{th} item respectively.
For example, if w was 4, we could either fit item 2 (w_2=3, v_2=9) or item 3 (w_3=2, v_3=5) into our knapsack so:

```
Value(4) = max {Value(4-3) + 9, Value(4-2) + 5}
         = max {Value(1) + 9, Value(2) + 5}
         = max {0 + 9, 5 + 5}
         = 10
```

Since we can't fit anything into a knapsack with a capacity of 1, `Value(1)` is 0 and the only option for capacity 2 is item 3 so `Value(2)`is 5. This gives a value of 10 for `Value(4)`. This partial example demonstrates a recursive top down approach.

For the full problem we can use a simpler bottom up approach which fills in the values of `Value()` using the following:

```
Value(0) = 0     With no capacity we have no items
for w = 1 to W (Knapsack capacity)
   Value(w) = max {Value(w-w_i) + v_i}     for all i such that w_i <= w
```

Then the solution to the problem is just `Value(W)`. We use a 1D array to store the values of `Value(w)`. For example, fill in the following array for the problem given on the previous page. Since W = 11, we need the following array of 12. Remember to start by putting a 0 in the first element. At each step if $w-w_i < 0$, we don't consider that item.

0	1	2	3	4	5	6	7	8	9	10	11

Filling in the array from left to right, you should get a value of 32 for `Value(11)`. If we need to find the actual items that were included to get the maximum value in our knapsack, we can backtrack through the array to see which items were included at each step. For example, starting with `Value(11)`, we go back to look at `Value(11-5)`, `Value(11-3)`, `Value(11-2)`and `Value(11-6)`. If $\text{Value}(w) = \text{Value}(w-w_i) + v_i$ then the i^{th} item must have been included at this step. Based on which item was included, we adjust the value of w accordingly and continue backtracking until w = 0.

The second type of knapsack problem doesn't allow repetition, so each item is used at most once. This is sometimes referred to as the 0/1 knapsack problem since an item must be put entirely in the knapsack or it is not included at all. Since we can't use an item more than once, the state of each stage of the problem is now described using two parameters – the current capacity `w` and the last item that we have considered `j`. When `j` = 3 that means we have considered items 1, 2 and 3. So we define:

`Value(w,j)` = maximum value using a knapsack with capacity `w` and items `1..j`

And the solution to our problem would be `Value(W, n)`. In this case, each items is either included or not. So we have:

$$\begin{aligned}\texttt{Value(w,j)} &= \texttt{max}\ \{\text{value when the } j^{th} \text{ item is included, value when the } j^{th} \text{ item is not included}\}\\ &= \texttt{max}\ \{\texttt{Value}(w-w_j, j-1) + v_j,\quad \texttt{Value}(w, j-1)\}\end{aligned}$$

Note: The first case is only valid if $w_j <= w$. Using a bottom up approach we need a 2D array to keep track of values of both `w` and `j`. We can fill in this table using the following steps:

```
Initialize all Value(0,j) = 0      With no capacity we have 0 value
And all Value(w, 0) = 0            Using no items we have 0 value
for j = 1 to n                     Consider each item
   for w = 1 to W                  Look at all possible capacities
      if $w_j$> w         $j^{th}$ item too large to include
         Value(w, j)= Value(w, j-1)
      else                pick the max value found by including the $j^{th}$ item or not
         Value(w, j) = max {Value(w-$w_j$, j-1) + $v_j$, Value(w, j-1)}
```

Solution is `Value(W, n)` - the best value with maximum capacity and all items included.

Try working through this algorithm with the earlier example and the following 2D array. The data is repeated on the right to save you turning pages. See the next page for more tips.

Item	Weight (kg)	Value ($)
1	5	14
2	3	9
3	2	5
4	6	17

	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					

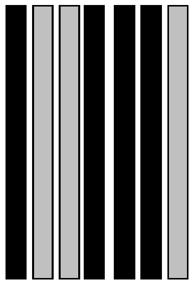
After filling in the top row and first column with 0's, you can fill in each column from top to bottom using the values in the column to the left (`j-1`) to figure out the next value. After filling in this table you should get a final value of 31 for `Value(11, 4)`. In this DP solution we are using the earlier solutions (results in the column to the left) to help find the solution to the next problem.

If we need to find the actual items that were included to get the maximum value in our knapsack, we can backtrack through the 2D array to see whether an item was included or not at each step. If $\texttt{Value(w, j) = Value(w-w}_{\texttt{j}}\texttt{, j-1) + v}_{\texttt{j}}$ then the $\texttt{j}^{\text{th}}$ item was included. Otherwise, it was not included. If the item was included, we need to adjust the value of `w`. We continue backtracking until all items have been considered.

In the above examples, we didn't need to completely fill our knapsack. However, in some variations of the knapsack problem you may need to fill the knapsack completely to capacity with no extra space. In these problems a solution is not guaranteed so you need to handle this possibility when solving the problem.

Dynamic Programming is a technique that can be used to speed up the processing time for a variety of problems. Before using DP, your first step is to recognize that DP is a valid approach to help solve the problem. If using DP, you need to identify how to define the state of each possible sub problem or stage. For example, in the second knapsack problem, the state of each stage (sub problem) was defined by the current capacity (`w`) and the items already used (`j`). After you have identified the sub problems (stages) and their state, the next step is to find a formula that relates the optimal solution of a larger problem (later stage) to the optimal solution of a smaller sub problem (earlier stage).

The following problem (from the UVa on-line judge site) can be solved using DP. Read over the problem carefully and then come up with the sub problems to consider, how to describe the state of each sub problem and a formula that relates the optimal solution of a larger sub problem to the optimal solution of a smaller sub problem.



A bar-code symbol consists of alternating dark and light bars, starting with a dark bar on the left. Each bar is a number of units wide. Shown on the right is a bar-code symbol consisting of 4 bars that extend over 1+2+3+1=7 units. In general, the bar code BC(n,k,m) is the set of all symbols with k bars that together extend over exactly n units, each bar being at most m units wide. For instance, the shown bar code belongs to BC(7,4,3) but not to BC(7,4,2). Below are all 16 symbols in BC(7,4,3). Each `1' represents a dark unit, each `0' a light unit.

```
0: 1000100 | 4: 1001110 | 8:  1100100 | 12: 1101110
1: 1000110 | 5: 1011000 | 9:  1100110 | 13: 1110010
2: 1001000 | 6: 1011100 | 10: 1101000 | 14: 1110100
3: 1001100 | 7: 1100010 | 11: 1101100 | 15: 1110110
```

Each input will contain three positive integers: n, k, and m (1 ≤ n, k, m ≤ 50). The output for each input will be the total number of symbols in BC(n,k,m). For example with input 7 4 3 the output will be 16 and for 7 4 2, the output will be 4.

4.6 Questions, Exercises and Problems

1) The following method solves the Towers of Hanoi puzzle. Fill in the blanks to complete this code. You can assume the towers would be numbered from 1 to 3.

```
static void moveRings (int noOfRings, int start, int end)
{
   if (____________________________________________________)
      System.out.println ("Move from: " + start + " to " + end);
   else
   {
      int intermediate = ____________________________________;
      moveRings (_________________________________________________);
      moveRings (_________________________________________________);
      moveRings (_________________________________________________);
   }
}
```

2) Given the following method:

```
     static int mystery (int number)
     {
         if (number < 10)
             return number;
         else
             return number % 10 + mystery (number /10);
     }
```

Predict the output of the following method calls.

```
      mystery(1234567)                mystery(987654)
```

What does the `mystery` method do?

3) The following `searchAndReplace` method uses recursion. Complete this code.

```
static String searchAndReplace (String str, String search, String replace)
{
  int findIndex = str.indexOf (search);
  if (_______________________________________)
  {
    StringBuilder strBld = new StringBuilder (__________________________);
    strBld.append (replace);
    strBld.append (searchAndReplace (__________________________________,
                                     search, replace));
    return strBld.toString();
  }
  else
    return ____________________________________;
}
```

4) A palindrome is a word or phrase that is the same forwards as it is backwards. For example, the following words and phrases are all palindromes:

```
Race carrot or race car?           radar
Sun at noon, tan us.               Dad
```

For the phrases, we ignore any spaces or punctuation when checking to see if the phrase is a palindrome. Also, in all of the examples, case doesn't matter so "Dad" should be considered a palindrome.

Complete the following code for a method that checks to see if a String is a palindrome using recursion. This recursive method has no loops.

```
static boolean isPalindrome(String str)
{
     // We must have a palindrome if ...
     if (_____________________________________)
          return true;

     // Get the first character and last character in the String
     // Also convert each character to lower case
     char firstChar = ____________________________________________;
     char lastChar  = ____________________________________________;

     // Deal with these special cases first to deal with non-letters
     if (!Character.isLetter(firstChar))
          return isPalindrome(____________________________________);
     if (!Character.isLetter(lastChar))
          return isPalindrome(____________________________________);

     // Check the next step
     if (____________________________________________________________)
          return isPalindrome(____________________________________);
     else
          return false;
}
```

5) If you want to write a method to find the factorial of a number, which is better: a recursive method or a non-recursive method? Explain your answer.

6) The CodingBat web site (`http://codingbat.com`) includes practice problems for recursion. Try completing the Recursion 1 and Recursion 2 sections on this site for practice. You can complete these problems without creating an account or logging in but if you log in with an account you can keep track of your progress.

To practice using recursion, solve Questions 7 to 10 using recursion without any loops. In each case, identify the base case and then try to think of how to simplify the problem in order to move closer to the base case.

7) Write a method called `charCount` that counts the number of times the given character appears in the given String. For example: `charCount("cats and dogs", 's')` should return 2. **Hint:** How many times would the given character appear in an empty String? Using this as your base case, you can get closer to the base case at each stage by reducing the length of the String by one.

```
/** Counts the number of times the given character
  * appears in the given String
  * @param str the String you are looking for characters in
  * @param ch the character you are counting in str
  * @return the number of times ch appears in str
  */
static int charCount (String str, char ch)
{









}
```

8) Write a method called `addStars` that adds `*`'s between all of the letters in the given String. For example: `addStars("RHHS")` should return `R*H*H*S`. To help think of a base case, think of a String that you would not need to add any `*`'s to.

```
/** Adds *'s between each of the letters in the given String
  * @param str the String that you want to add the *'s to
  * @return a String with *'s between each letter
  */
static String addStars (String str)
{






}
```

9) Write a method called `arraySum` that finds and returns the sum of all of the elements in a given array. For example, given `numbers = {8, 4, 5}`, then `arraySum(numbers, 0)` should return 17. Since we can't easily create a sub array of an array (unlike substrings), we have added a second parameter to indicate the index of the starting point of the sub array that we want to consider in each recursive call. Since the initial call wants to add up all of elements of the array, we would initially pass in an index of 0.

```
/** Finds the sum of all of the elements of the given array
  * from the given start index to the end of the array
  * @param numbers the array of integers to find the sum of
  * @param start the start index of the sub array to include in the total
  * @return the sum of the elements from the start index to the end
  */
static int arraySum (int [] numbers, int start)
{




}
```

10) Given an array of integers and a target total T, you want to check if it is possible to find a group of numbers from the array that adds up exactly to T. The size of the group can range from 0 to the length of the array. Write the code for a `boolean` method called `isSumPossible` that checks to see if the given total is possible using numbers from the given array. Just like question 9, at each recursive call we only consider part of the array starting with the given start index until the end of the array. **Hint 1:** At each step you have two choices. In one case you include the next number (`numbers[start]`) in your total and in the other case you don't. If you include the number, don't forget to update the target total accordingly. **Hint 2:** For what value of target can you return `true` immediately?

```
/** Checks to see if we can use a group of numbers from the given
start
  * index to the end of the array to sum exactly to the given total
  * @param numbers the array of integers to use to make the total
  * @param target the total that we are trying to get
  * @param start the start index of the sub array to use to make the total
  * @return true if we can get the exact total using the given
  *         sub array, false if it is not possible
  */
static boolean isSumPossible (int [] numbers, int target, int start)
{




}
```

11) The following questions refer to the `isSumPossible` method presented previously.

a) if `numbers = {-5, 9, 7, -3, 19, 4}`, what will be the return value of the following? **Hint:** If you remember what the `isSumPossible` method does you don't need to trace all of the steps of the recursion to solve this problem.

i) `isSumPossible (numbers, 13, 0)`

ii) `isSumPossible (numbers, 6, 2)`

iii) `isSumPossible (numbers, 39, 1)`

b) If we add the following code as the first line of the `isSumPossible` method:

```
System.out.println(target + ", " + start);
```

What will be the output produced by this new `println` statement inside `isSumPossible` for each of the following?

i) Given `numbers = {3, 7, 6}`, and you call: `isSumPossible (numbers, 9, 0)`
ii) Given `numbers = {4, -2, 8, 7}`, and you call: `isSumPossible (numbers, 5, 0)`

You may want to draw the resulting tree structure for the recursion for each question

12) What does it mean if a sorting algorithm is stable? Considering the Selection Sort, Insertion Sort and Merge Sort, which of these sorts are stable? For each of the non-stable sorts, clearly explain why it is not stable (give a specific counter example to show that the sort is not stable).

13) The Java API built in sort (`Arrays.sort()`) uses a tuned quick sort to sort primitive types (characters, integers and doubles) and a modified merge sort to sort objects. Why do you think it uses two different sorting algorithms depending on what it is sorting?

14) Looking at the algorithm for the Selection Sort, how many element comparisons are needed for this sort in the best, worst and average case? Also, how many swaps are needed? What is the runtime in Big O notation of this sort? Explain your answers.

15) Looking at the algorithm for the Insertion Sort, how many element comparisons are needed for this sort in the best, worst and average case? Also, how many data movements are needed? What is the runtime in Big O notation of this sort? Explain your answers.

16) Comparing the Selection Sort and the Insertion Sort, which sort do you think will be faster? Clearly explain your answer.

17) Looking at the algorithm for the Merge Sort, how many element comparisons are needed for this sort? Also, how many data movements are needed? What is the runtime in Big O notation of this sort? Explain your answers.

18) In section 4.4 we looked at three different types of sorts. Other popular sorting algorithms include the Quick Sort, Shell Sort, Heap Sort and Radix Sort. Write a method that uses one of these sorting algorithms to sort an `ArrayList` of `Strings`.

19) In the following exercise we will be looking at the code and the run-time for the three (3) different sorts covered in this chapter under various conditions.

The program `SortComparison.java` contains the code for a Selection Sort, Insertion Sort and Merge Sort given in section 4.4. It also contains a main program and some other methods to help test the run time of these sorts.

The first part of the main program checks that each sort works properly. If you change any of the code in these sorts, make sure that they are still correct after making the changes. Remember the first priority of any program is that it works!

Using the 3 given sorting methods, complete the following table. To be consistent you should time your algorithm using the classroom computers. In each case run the program a few times and use the average time. Times should be given in milliseconds.

Sorting Times (in ms) for Different Size ArrayLists – **Selection Sort**

Original Order of the List	10,000 element List	20,000 element List
Random Order		
Sorted Order		
Reverse Order		

Sorting Times (in ms) for Different Size ArrayLists – **Insertion Sort**

Original Order of the List	10,000 element List	20,000 element List
Random Order		
Sorted Order		
Reverse Order		

Sorting Times (in ms) for Different Size ArrayLists – **Merge Sort**

Original Order of the List	10,000 element List	20,000 element List	100,000 element List	1,000,000 element List
Random Order				
Sorted Order				
Reverse Order				

20) Complete the analysis given above for the sort you picked in Question 18. Use the chart used for the Merge Sort in your analysis.

21) For each of the three sorts covered in question 19, clearly explain why the order of the list you want to sort (ordered list or reverse ordered list) is faster, slower or the same speed as the random ordered list.

22) Assume you want to sort the following list of numbers: `16 8 12 2 14 4 6 10`

a) If you used a **Selection Sort** to sort this list, how many **swaps** would be required? Give the exact number of swaps for this particular list. Show your work.

b) If you used an **Insertion Sort** to sort this list how many element **comparisons** would be required? Give the exact number of comparisons for this particular list. Show your work.

23) Looking at the code for the Selection Sort, why is the upper limit for the outer loop `list.size() – 1`? Explain your answer.

24) Looking at the code for the Insertion Sort, if you changed the inner `while` loop to:

```
while (index >= 0 && list.get(index).compareTo(insert) >= 0)
```

would the sort still work? Explain your answer.

25) In the Insertion Sort, if you used a binary search to find the point of insertion of each element you are inserting could you speed up the run time of this sort? Try updating the code for the insertion sort to see if your answer is correct.

26) Looking at the code for the Merge Sort:
a) If you changed the first `if` statement in the `merge` method to the following:

```
   if (list.get(middle).compareTo(list.get(middle - 1)) > 0)
      return;
```

would the sort still work? Explain your answer.

b) If you changed the first `if` statement in the first `while` loop of the `merge` method to the following:

```
if (list.get(firstIndex).compareTo(list.get(secondIndex)) < 0)
```

would the sort still work? Explain your answer.

27) If a $O(n \log n)$ sort takes 1.5 milliseconds to sort a List of 600 elements, approximately how long would it take to sort a similar List of 10,000 elements? Show your work.

28) Write a Java program to solve the Longest Common Subsequence problem. In this problem you are given two strings (e.g. DXNYA and YDNCXA) and you want to find the longest sequence of characters that appear in left to right order (but not necessarily beside each other) in both strings. For the given example strings, the answer would be DNA. This problem is a variation of the Edit Distance problem which, like this problem, can be solving using dynamic programming. **Hint:** The state of each sub problem can be described by the current positions in each of the strings.

29) In the game of Boggle you randomly mix up some letter cubes to get a grid of letters. On the right is an example of a 4 by 4 Boggle board:

S	E	R	S
P	**A**	**T**	G
L	I	**N**	**E**
S	E	**R**	**S**

The objective of the game is to find as many words as possible on the given board in a limited amount of time. A word can be made by moving from letter to letter horizontally, vertically or diagonally. However, in any given word, each letter on the board can only be used once. For example, on the board given above, the word PLANTERS can be found by starting with the P on the left and then tracing out the letters by moving down (L), to the upper right (A), lower right (N), up (T), lower right (E), lower left (R) and then right (S).

How many words (with at least 3 letters) can you find on this board? Make a note of what you are doing when you are finding words.

The score for each round is based on the words that you found. The points for each word on the board are determined by their length (1 point for words of 3 or 4 letters, 2 points for 5 letter words, 3 points for 6 letter words, 5 points for 7 letter words and 11 points for any words with 8 or more letters). For a 5x5 or larger grid, 3 letter words are not allowed so 1 point is awarded for 4 letter words only. What do you think is the maximum possible score for the Boggle Board given above?

To create the game of Boggle, we will start with a `BoggleBoard` class. As a minimum your new class should have the following methods:

public `BoggleBoard (String filename)` – constructs a new `BoggleBoard` from the data in a file with the given name. Each boggle data file will contain either 4 or 5 lines of text to represent either a 4 x 4 or 5 x 5 grid of letters. An example file is shown on the right. **Hint:** See Appendix G.

```
SERS
PATG
LINE
SERS
```

public boolean `search(String word)` – looks for a word in the letter grid using the Boggle rules, returning true if the word is found. You should write another private recursive helper method (e.g. `searchAround(row, col, wordLeft)`) to complete this task.

public int `getMaxScore (List<String> wordList)-` finds the maximum score of this `BoggleBoard,` based on the rules of the game of Boggle. This method would have a single parameter that would consist of a list of valid words to search for. You can call the `search` method written above in this method.

public `String toString ()` – returns a String representation of this `BoggleBoard`. This method could be used for debugging purposes.

You should also write a main program to test your methods. This program should create a `BoggleBoard` object from a file, display this board and then find and display this board's maximum score.

30) Write a program to find the shortest path through a maze. Each maze is defined by a 2D array of characters. The characters 'S', 'F', 'X' and '-' indicate the starting position, finishing position, a maze wall and an open path respectively. The 'S' will always be in the first column. A sample maze is shown to the right.

```
S-----XXXX
XX-XX----X
X-----XX-X
X-XXXXXX-X
X--------F
```

To move through the maze you can move up, down, left or right. You <u>cannot move diagonally</u>. For each maze you want to find the fewest number of moves required to get from the start to the finish. For example, in the above maze we can travel from S to F in 13 moves. Mazes should be input from a file. The first line of the file indicates the number of mazes to evaluate. For each maze, the next line indicates the size of the maze in rows and columns (separated by a space) followed by the maze of characters. A sample input file is given below:

```
3
5 10
S-----XXXX
XX-XX----X
X-----XX-X
X-XXXXXX-X
X--------F
12 15
S-------XXXXXXX
XXXXX-XXXX----X
X---X---X--XX-X
X-XXX-X---XX--X
X-X---XXXX----X
X-X-XX-----XXXX
X------XXX----X
X-XX--X-X----XX
X----XX----XXXX
XX-X--XXXXXF---
XX-XX-X---XXXX-
X---X----------
2 3
SX-
-XF
```

For each maze, your program should output the <u>minimum</u> number of moves required to get from S to F. In some mazes there may be no solution. Output for the above set of mazes would be:

```
Solving Mazes Program
Maze 1: 13 moves
Maze 2: 34 moves
Maze 3: No solution
Maze program is complete
```

Once again you should plan out your program carefully before beginning. You should also create a `Maze` object to keep track of each maze. Note: To solve all of the given test cases, in a reasonable time, your code will need to be efficient. See Appendix G for some tips on reading in the Maze file.

The following problem is based on a problem from the Canadian Computing Competition

31) Roberta the Robot plays a perfect game of golf. When she hits the golf ball it always goes directly towards the hole on the green, and she always hits exactly the distance that is specified for the club. Each such action is known as a stroke, and the object of golf is to hit the ball from the tee to the hole in the fewest number of strokes. Roberta needs a program to select the best combination of clubs to reach the hole in the fewest strokes.

She also needs to decide if the task is impossible, in which case she graciously acknowledges the loss. Roberta can carry up to 32 clubs, and the total distance from the tee to the hole does not exceed 5280 metres.

The Input

The first line of input gives the number of test cases. For each test case, you are given the distance from the tee to the hole, an integral number of metres between 1 and 5280. The next line states the number of clubs, between 1 and 32. This will be followed by a line containing the distances (in metres) that each club will hit the ball, an integer between 1 and 100. Each club distance will be separated by a single space and no two clubs will have the same distance.

The Output

For each test case, if Roberta can get the ball from the tee to the hole, without going pass the hole, print "Roberta wins in *n* strokes." where *n* is minimized. If Roberta cannot get the ball from the tee to the hole, print "Roberta acknowledges defeat.".

Sample Input (from a text file)

```
2
101
3
50 21 20
101
8
54 19 70 69 33 27 91 77
```

Sample Output (to the screen**)**

```
Roberta wins in 5 strokes.
Roberta acknowledges defeat.
```

The following problem is based on a problem from the UVa on-line judge

32) There is a SuperSale in a SuperHiperMarket. Every person can take only one object of each kind, i.e. one TV, one carrot, but for an extra low price. We are going with a whole family to that SuperHiperMarket. Every person can take as many objects, as he/she can carry out from the SuperSale. We are given a list of objects with prices and their weight. We also know the maximum weight that every person can carry. We need to find the maximum value of objects that we can buy at the SuperSale?

Input Specification

The input consists of *T* test cases. The number of them (1<=*T*<=*1000*) is given on the first line of the input file.

Each test case begins with a line containing a single integer number *N* that indicates the number of objects (*1 <= N <= 1000*). Then follows *N* lines, each containing two integers: P and W. The first integer (1<=P<=100) corresponds to the price of the object. The second integer (1<=W<=30) corresponds to the weight of the object. The next line contains one integer (1<=G<=100) indicating the number of people in our family group. The next G lines contains maximal weight (1<=MW<=30) that the i*th* person from our family (1<=i<=G) can carry.

Output Specification

For every test case your program has to determine one integer. Print out the maximal value of goods which we can buy with that family.

Sample Input

```
2
3
72 17
44 23
31 24
1
26
6
64 26
85 22
52 4
99 18
39 13
54 9
4
23
20
20
26
```

Output for the Sample Input

```
72
514
```

The following problem is based on an USACO problem by Ray Li

33) Farmer John wants to remodel the floor of his barn using a collection of square tiles he recently purchased from the local square mart store (which of course, only sells square objects). Unfortunately, he didn't measure the size of the barn properly before making his purchase, so now he needs to exchange some of his tiles for new square tiles of different sizes.

The N square tiles previously purchased by FJ have side lengths $A_1...A_N$. He would like to exchange some of these with new square tiles so that the total sum of the areas of the tiles is exactly M. Square mart is currently offering a special deal: a tile of side length A_i can be exchanged for a new tile of side length B_i for a cost of $|A_i-B_i|*|A_i-B_i|$ units. However, this deal only applies to previously purchased tiles so FJ is not allowed to exchange a tile that he has already obtained via exchanging some other tile (i.e., a size 3 tile cannot be exchanged for a size 2 tile, which is then exchanged for a size 1 tile).

Please determine the minimum amount of money required to exchange tiles so that the sum of the areas of the tiles becomes M. Output -1 if it is impossible to obtain an area of M.

INPUT FORMAT: (For each test case – the test file will contain 10 test cases)

- Line 1: Two space-separated integers, N (1<=N<=10) and M (1<=M<=10,000).
- Lines 2..1+N: Each line contains one of the integers A_1 through A_N, describing the side length of an input square (1<=A_i<=100).

SAMPLE INPUT: (For one test case)

```
3 6
3
3
1
```

INPUT DETAILS:
There are 3 tiles. Two are squares of side length 3, and one is a square with side length 1. We would like to exchange these to make a total area of 6.

OUTPUT FORMAT:

- Line 1: The minimum cost of exchanging tiles to obtain M units of total area, or -1 if this is impossible.

SAMPLE OUTPUT: (For one test case)

```
5
```

OUTPUT DETAILS:
Exchange one of the side-3 squares for a side-2 square, and another side-3 square for a side-1 square. This gives the desired area of 4+1+1=6 and costs 4+1=5 units.

Chapter 5 – Working with Numbers in Java

This chapter looks at how both integer and real numbers are stored in Java. It also includes information about working with `Integer` and `Double` objects as well as `BigInteger` objects for larger integers.

5.1 Number Systems (Decimal, Binary and Hexadecimal)

Decimal (Base 10) numbers have 10 digits (0 to 9) with the following places:

Place	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}
Value	1000	100	10	1		1/10	1/100
Number	7	3	4	5	.	7	8

Each digit in the number tells you the quantity of the corresponding place value. In our example we have 7 1000's, 3 100's, 4 10's, 5 1's, 7 1/10's and 8 1/100's.

Binary (Base 2) numbers have 2 digits (0 or 1) with the following places:

Place	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}
Value	8	4	2	1		1/2	1/4
Number	1	0	1	0	.	1	1

To find the decimal value of a binary number, we add up the place values. In the above example, we have 1×8, 1×2, 1×½ and 1×¼, for a total of 10.75. Since there are only 2 possible binary digits (0 or 1), each place value is either included or not.

Here are two more examples. In each case, you should write the binary place values on top of the given digits and then add up the total of all of the place values with a 1 digit. Convert the following binary numbers to decimal:

```
1 0 1 1 0 0 1 . 1                1 0 1 1 1 0 1 1 0 0 . 0 0 1
```

To convert a decimal number to binary, we need to determine which binary place values we need to include to make up the given number. For example, if we want to convert 87.25 to binary, we need to include the place values shown below. Starting with the 1 place, we write down the place values to the left doubling the value each time (2, 4, 8 ...). We stop at 64 since there will be no 128's in 87.25. To find the place values to the right, we half the value each time (½, ¼ ...).

64	32	16	8	4	2	1		½	¼
1	0	0	1	1	0	1	.	0	1

As we fill in each place with a 1 or 0 we subtract out the value of the place. For example we have 1×64 so 87.25 – 64 = 13.25 then we have 0×32's and 0×16's and 1×8 so 13.25- 8 = 5.25 and then we have 1×4, 1×1 and 1×¼ to make up the rest of the number. The final result is shown above.

In the previous example, to find the value on the right hand side of the binary place was quite easy since the fractional part was 0.25. However, in some cases it can become quite difficult to find the fractional part in binary. For example, what about 0.3?

For 0.3 we could subtract 0.25 to get 0.05 then try 0.125 and 0.0625 etc., but there is an easier way. We start with the fractional part (see column on the right) and then we double it. We then take the fractional part of the doubled value and we keep doubling it. Remember, at each step we are only doubling the fractional part. For example with 1.2 we only double the 0.2.

```
0.3
___
0.6
___
1.2
0.4
0.8
1.6
___
1.2
0.4
0.8
1.6
```

Eventually we stop when we get 0 (a non-repeating binary number) or we get a repeating pattern which means we have a repeating binary number. For example, starting with 0.3 we eventually see a pattern emerge that will keep going on forever. To get the fractional bit pattern of the binary number we look at the 0's or 1's on the left (not including the original 0 digit in the starting number) reading from top to bottom. So we take the `0` from `0.6`, the `1` from `1.2`, the `0` from `0.4`, the `0` from `0.8` and finally the `1` from `1.6`. Putting these digits together from left to right after the binary place we get 0.3 in binary will be: $0.0\overline{1001}$. The bar over the last 4 digits indicates the repeating digits.

Since 0.3 in decimal becomes a repeating binary number with an infinite number of binary digits, this number cannot be stored accurately in a memory location with a finite number of bits. We will talk more about this in section 5.3 when we discuss how real numbers are stored in Java. In the meantime, try converting 0.625 and 0.85 to binary on your own using the technique given above.

Hexadecimal and Binary Numbers

Since each 4 bit binary number represents a value from 0 to 15, hexadecimal numbers (base 16) are sometimes used to represent larger binary numbers. Since hexadecimal numbers need 16 single digits, the letters A through F are used to represent the values 10 to 15. Here are the 16 hex digits and their corresponding 4 bit binary equivalent:

Hex Digit	4 –Digit Binary	Hex Digit	4 –Digit Binary	Hex Digit	4 –Digit Binary	Hex Digit	4 –Digit Binary
0	0000	4	0100	8	1000	C (12)	1100
1	0001	5	0101	9	1001	D (13)	1101
2	0010	6	0110	A (10)	1010	E (14)	1110
3	0011	7	0111	B (11)	1011	F (15)	1111

If you want the bit pattern of the last 8 bits in an `int` to be 10100101 we can use the combined 4 bit patterns `1010` – A and `0101` – 5 to give A5. In Java we can indicate that a number is a hexadecimal number by including the `0x` prefix. For example:

```
int number = 0xA5;  // would store 00000000000000000000000010100101
```

In the above example, the 0's on the left are not needed. How would you store the bit pattern: `00011111101101110011001011011001` in a Java `int` variable?

5.2 Storing Integer Numbers in Java

In Java, there are 4 different types of integer primitives. These types are summed up in the table below.

Type	Number of Bits	Minimum Value	Maximum Value
`byte`	8	-2^{7} (-128)	$2^{7}-1$ (127)
`short`	16	-2^{15} (-32,768)	$2^{15}-1$ (32,767)
`int`	32	-2^{31} ($\sim -2\times10^{9}$)	$2^{31}-1$ ($\sim 2\times10^{9}$)
`long`	64	-2^{63} ($\sim -9\times10^{18}$)	$2^{63}-1$ ($\sim 9\times10^{18}$)

Each integer type includes a left most sign bit. Therefore the range of values for each type is -2^{n-1} to $2^{n-1}-1$ where n is the number of bits. For example, given:

`byte` `first = 78;` - the memory bit pattern is:

0	1	0	0	1	1	1	0

To store a negative value it may seem logical to use the same bit pattern with a 1 in the sign bit to indicate that the number is negative. Although this looks like a practical way to us, this method doesn't work very well for the computer. Therefore, most computers use a method called "two's complement" to store integer numbers including negative numbers. To find the bit pattern of negative numbers (such as -38) using "two's complement", we do the following. Note: This example is for a `byte` variable.

1) Find the 8-bit (for a byte) bit pattern of the positive value of the number.
2) Complement (flip) the bits in this number (change all of the 0's to 1's and 1's to 0's).
3) Add 1 to the number. When adding 1 to a binary number, don't forget to carry.

For example, for **`byte`** `second = -38` we get

0	0	1	0	0	1	1	0	Positive value of the number (e.g. 38)
1	1	0	1	1	0	0	1	Flip all of the bits
1	1	0	1	1	0	1	0	Add 1 – Bit pattern for -38 in memory

If the sign bit is set in the bit pattern of number, we know the number must be negative and we can do the same flipping of bits and adding 1 to find the original negative number. For example, if we were given the following bit pattern `10101011` in a `byte` variable we first notice that the sign bit is set so it must be a negative number. Since the number is negative, we must complete the same steps given above.

1	0	1	0	1	0	1	1	Sign bit set – this number is negative
0	1	0	1	0	1	0	0	Flip the bits
0	1	0	1	0	1	0	1	Add 1 – Result is 85 so original number was -85

Therefore, the bit pattern `10101011` in a `byte` is for -85. If the sign bit is `0`, we just convert the number to decimal in the normal way. Here are some more examples.

- Find the bit pattern for -95 if it was stored in a `byte` variable.
- Given the bit pattern for a `byte` was `10111001`, what is the number?
- Given the bit pattern for a `byte` was `00111111`, what is the number?

To show the advantages of using “two’s complement”, let’s add the two values we just found. Remember when adding bits we get lots of carry bits. Note: The left most carry bit will be lost since a byte can only hold 8 bits.

0	1	0	0	1	1	1	0	Bit pattern for 78
1	1	0	1	1	0	1	0	Bit pattern for -38
0	0	1	0	1	0	0	0	Add up the bit patterns – we get 40

So using “two’s complement” makes it very easy for the computer to add numbers even when the numbers are negative.

Promotion and Casting of Integers

When you assign a smaller integer (e.g. `byte`) to a larger integer (e.g. `short`), the number is promoted up to the larger integer. During this promotion process, how do we fill in the extra bits? Using the example from the previous page, if we assign the value in the `byte` variable `first` (78) to a `short`, we have 8 extra bits to fill. Given

`short` `newFirst = first;` - the bit pattern for `newFirst` will be:

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0	1	0	0	1	1	1	0

In this case, the 8 new bits were filled with 0’s - the bit value of the sign bit of the original `byte` variable. This process is known as “sign extension” and it makes the promotion work, since the bit pattern for 78 in the `short` is correct. But what happens with negative numbers? Let’s try promoting the `second` variable (-38) to a `short`.

`short` `newSecond = second;` - the bit pattern for `newSecond` will be:

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	1	1	0	1	1	0	1	0

Following the “sign extension” rule, the 8 new bits are filled with 1’s since the sign bit of the original `byte` variable was “1”. Once again, we can see that the bit pattern is correct (if you are not sure, try flipping the bits and adding 1 for the new `short` number) proving that “sign extension” works for both positive and negative numbers.

Now let’s see what happens when we **cast** an integer variable. Given

`short` `third = 823; -` the bit pattern for `third` will be:

0	0	0	0	0	0	1	1	0	0	1	1	0	1	1	1

Then if we cast this value to a `byte` using

`byte` `newThird = (`**`byte`**`) third; -` the bit pattern for `newThird` will be:

0	0	1	1	0	1	1	1

In this case, the 8 bits on the left are lost and the number stored in the `byte` will be 55 not 823. Since we may lose information during casting we need to be careful. In some cases, casting can change a positive number to a negative number if the sign bit in the new number ends up being 1 after discarding the left most bits.

Integer Overflow

Since integer variables in Java have a finite number of bits, if a number gets too large for the number of allocated bits we get an integer overflow and information is lost just like when we cast. For example, predict the output of the following:

```
byte number = 127;
number +=3;
System.out.println(number);
```

If number was an `int` or even a `short` and we ran this code, we would get the expected output of 130. However, if we try to store 130 in a `byte`, we find that, even though all of the bits fit, the left most bit is now the sign bit so we get a negative result.

1	0	0	0	0	0	1	0	Bit pattern for 130 – will be negative in a `byte`
0	1	1	1	1	1	0	1	So we need to flip the bits
0	1	1	1	1	1	1	0	Add 1 to get 126 so original number was -126

Therefore, the above code would output -126. Since the maximum value we can store in a byte is 127, when we tried to add 3 to 127 the value overflowed and we ended up with -126. We got the same results in Grade 11 when we were calculating factorials that were too large to be stored in an `int` or even a `long` in some cases.

In addition to positive overflow we can also get negative overflow if we try to go below the minimum value that can be stored in an integer. For example, try changing the initial value of `number` to `-128` and the calculation to `number-=3`. In this case the result will be +125.

5.3 Storing Real Numbers in Java

In Java, we have two primitive types to store real numbers, a 32-bit `float` and a 64-bit `double`. In Java, real numbers are stored using the IEEE 754 standard. Here is an example.

```
float number = 37.85;
```

Converting `37.85` to binary we get $100101.11\overline{0110}$ or $1.0010111\overline{0110} \times 2^5$. Notice that we use a base of 2 for our exponent since everything in the computer is in binary. Now to store this `float` variable we need to review what the 32-bits of the float are used for. The first bit is the sign bit, the next 8 bits are the exponent and the last 23 bits are for the significand. Storing `37.85` in a `float` would give us:

s	Exponent (+ bias of 127)								Significand (without the left most 1)																						
0	1	0	0	0	0	1	0	0	0	0	1	0	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0

Since the number is positive the first bit (sign bit) is 0. To store the exponent of 5 we first add a bias of 127 so 5 + 127 = 132 or `10000100` in binary. The value of the actual exponent can be between -126 and +127 which would be stored as a value between 1 and 254 after adding the 127. This bias is used for exponents instead of "two's complement" in order to make it easier to compare two `float` variables.

Finally, for the significand we store the digits in our number: $1.0010111\overline{0110}$ without the left most 1 (i.e. $0010111\overline{0110}$). Since the left most digit of our significand is always 1 (if it wasn't 1, we would change the exponent) it doesn't need to be stored. By not storing this leading 1 we get 1 more binary digit of accuracy. This gives us 24 bits of accuracy in binary which translates into around 7 digits of accuracy in the corresponding decimal number.

The final result is shown on the previous page. As was mentioned earlier, since this number was a repeating binary number some of the repeating digits are not stored so the number 37.85 will not be stored exactly in a `float` variable.

Since the order of the `float`'s components from left to right are the sign, exponent and then the significand and because float's do not use "two's complement" for the exponent, when comparing two `float` variables we only need to compare the `int` equivalents of each `float`'s bit pattern. This results in quicker comparisons for `floats`. However, you should be careful when comparing `floats` for equality since real numbers may not be stored exactly so values that you would expect to be equal may not be equal.

Given the 32-bit memory bit pattern for a float, let's try to convert this back to a decimal value. For example, given:

s	exponent								significand																						
1	1	0	0	0	1	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Let's work through this step by step.

1) First we see that the sign bit is 1, so this number must be negative.

2) Then we look at the value in the exponent bits which is `10001001` or 137 in decimal. So taking away the bias of 127, the actual exponent is 10.

3) The significand is `111101110111` or **1**`111101110111` when we add back the left most 1 that wasn't stored earlier. Therefore our final number is:

$-1.111101110111\times 2^{10}$ = `-11111011101.11` which becomes `-2013.75`

A similar process is used to store `double` variables except we have more bits to store larger and more accurate numbers. With 64 bits we get 1 sign bit, 11 bits for the exponent with a bias of 1023 and a range of values between -1022 and +1023 and 52 bits for the significand with 53 bits of accuracy or approximately 15 decimal digits.

You may have noticed that the range of exponents presented earlier didn't include 0 or the maximum exponent (255 for `floats` or 2047 for `doubles`). These extreme exponents are used to handle some special numbers such as 0, +/- infinity and NaN (not a number). The following table summarizes these special values.

Exponent	Significand	Resulting Number
0	0	0 (Needs a special case since significand drops the leading 1)
255 or 2047	0	+/- infinity based on the sign bit
255 or 2047	Non-zero	NaN (Not a Number). (e.g. $\sqrt{-1}$)

5.4 Bit Wise Operations (Optional)

When working with integer numbers, Java includes a few bit wise operators that can be used to manipulate bits. Here is a summary of these operators with some examples:

Name	Symbol	Description	Example

Bitwise complement NOT	~	Inverts a bit pattern making every 0 a 1 and every 1 a 0	`byte one = 43; // 00101011`
			`byte two = (byte)~one;`
			would set `two` to `-44` (`11010100`)

Signed left shift	<<	Shifts a number by the given number of positions to the left	`int one = 43; // 00101011`
			`int two = one << 2;`
			would set `two` to `172` (`10101100`)

Signed right shift	>>	Shifts a number by the given number of positions to the right. Left most position is determined by sign extension.	`int one = -114;`
			`11111111111111111111111110001110`
			`int two = one >> 3;`
			would set `two` to `-15`
			`11111111111111111111111111110001`

Unsigned right shift	>>>	Shifts a number by the given number of positions to the right. Shifts a 0 into the left most position.	`int one = -114;`
			`11111111111111111111111110001110`
			`int two = one >>> 3;`
			would set `two` to `536870897`
			`00011111111111111111111111110001`

Bitwise AND	&	Performs a bitwise AND
		0 & 0 = 0 0 & 1 = 0
		1 & 0 = 0 1 & 1 = 1
			`  10100101`
			`& 11010111`
			`= 10000101`

Bitwise Exclusive OR	^	Performs a bitwise XOR
		0 & 0 = 0 0 & 1 = 1
		1 & 0 = 1 1 & 1 = 0
			`  10100101`
			`^ 11010111`
			`= 01110010`

Bitwise OR	|	Performs a bitwise OR
		0 & 0 = 0 0 & 1 = 1
		1 & 0 = 1 1 & 1 = 1
			`  10100101`
			`| 11010111`
			`= 11110111`

Generally, using bit wise operators will make your code more cryptic and harder to follow. Therefore, in most cases, you should avoid using these operators. However, in some situations (especially in some contest questions) using bitwise operators can be very useful and will sometimes make your code faster. Here are some examples. Since bitwise operators have very low precedence, the parentheses shown are required.

Using `1 << n` instead of `Math.pow(2, n)` is a faster way of getting a power of 2. For example, `1 << 7` is the same as 2^7.

We can use the right shift operator (>>) and a bit mask (&) to break a 32 bit integer into smaller components. For example to get the value of the left most 8 bits of a 32 bit int number (alpha value of a 32 bit colour value), we could use:

```
int alpha = (colour >> 24) & 0xFF;
```

The expression: `number & -number` will give the value of the right most bit of `number`. For example, for 12 (1100) the value of the right most bit will be 4. You can prove this by remembering how two's complement stores negative numbers and looking at the resulting bit patterns (see question 11 at the end of the chapter). We can use this result to check if a number is a power of 2. If `number` is a power of 2 then the boolean expression `(number & -number) == number` will be true.

Another interesting application that involves using bitwise operators is to use a single `int` variable to keep track of the items in a set (from Competitive Programming 3). In this case we would use each bit to keep track of whether the item is in the set (1) or not in the set (0). This can be a little bit confusing but it is a time and space efficient way of doing this. To avoid problems with the sign bit and two's complement, this technique can be used to track a set of up to 30 items using an `int` and 62 items using a `long`.

For example, assuming we use a variable called `set` to keep track of the items in a set and using zero based indexing starting from the right, if `set=37` or `100101` in binary then items 0, 2 and 5 would be included in this set. If we had a set that contained items 1, 3 and 6 the resulting bit pattern would be: `1001010` and the corresponding value of `set` would be: 74. To use this value to check on items in the set, we can use the following operations:

In the following operations, `(1<<j)` gives you a binary number with the `j`th bit set to 1 and all other bits 0. For example the 8 right most bits of `1<<5` would be `00100000`. This value can be used to look at or change the status of the `j`th bit in `set`. Remember, we are using 0 based indexing starting from the right so the right most bit is the 0th bit.

To set/turn on the `j`th item of the set, use the bitwise OR operation:
`set |= (1<<j)`.

To check if the `j`th item is included in the set, use the bitwise AND operation:
`result = set & (1<<j)`. If the `result != 0`, then the `j`th item is in the set and if `result = 0`, the `j`th is not in the set.

To clear/turn off the `j`th item of the set, use the bitwise NOT and AND operations:
`set &= ~(1<<j)`.

To toggle (flip the status) off the `j`th item of the set, use the bitwise XOR operation: `set ^= (1<<j)`.

To turn on all of the bits in a set of size n we can use:
`allBits = (1 << n) - 1`

Using an `int` variable and the ideas presented above we can look at all possible combinations in a set by going through a simple `for` loop. This is shown in the following example that is a variation of the `isSumPossible` method presented in Question 10 in Chapter 4. It is a bit slower than the recursive version presented earlier but it returns the array of numbers that make up the target total instead of just true or false.

```
/**
 * Finds and returns an array of the numbers in the given array that
 * can be added to get the target total. Returns null if not possible
 * @param numbers the array of integers to use to make the total
 * @param target the total that we are trying to get
 * @return an array of all of the numbers in numbers that can be added
 *         to make the target total or null if target is not possible
 */
static int [] numbersInSum (int[] numbers, int target)
{
   // Find the bit pattern when all items of the set are included
   // For example, for n = 7 this will be 1111111
   int allItems = (1 << numbers.length) - 1;

   // Look at each combination (e.g. 00000 to 11111)
   for (int set = 0; set <= allItems; set++)
   {
      // Find the total for the items in this combination
      int total = 0;
      for (int index = 0; index < numbers.length; index ++)
      {
         // Add the items that are in this set
         if ((set & (1 << index)) != 0)
            total += numbers[index];
      }
      if (total == target)
      {
         // Create and return an array of all of the items that
         // added up to the target total.  Uses Integer.bitCount
         // to find the number of elements in the new array
         int newIndex = 0;
         int [] setOfNumbers = new int[Integer.bitCount(set)];
         for (int index = 0; index < numbers.length; index++)
            if ((set & (1 << index)) != 0)
               setOfNumbers[newIndex++] = numbers[index];
         return setOfNumbers;
      }
   }
   // If no combination met the target
   return null;
}
```

Java includes some static methods in the `Integer` class that may be useful when working with the bits in an `int`. See the API help for more details on these methods. `Integer.bitCount` was used in the above example.

```
Integer.highestOneBit(int number)
Integer.lowestOneBit(int number)
Integer.numberOfTrailingZeros(int number)
Integer.numberOfLeadingZeros(int number)
Integer.bitCount(int number)
```

These methods are also available in the `Long` class. For `long`s you need to change the class name in front (e.g. `Long.bitCount(`**`long`**` number)`).

5.5 The `Integer` and `Double` Classes

In most cases the Java primitive types should be your first choice to store numbers. However, as we saw in section 2.4, if we want an `ArrayList` of numbers we will need a number object, since `ArrayLists` can only store objects. To handle this, Java includes a few “wrapper” classes to store number objects.

The `Integer` class wraps a value of primitive type `int` in an object. It contains a single field whose type is `int`. In addition, this class provides several methods and some constants that can be useful when dealing with integers. Here is a summary of some of these methods:

`Integer(`**`int`**` value)` - Constructs a newly allocated `Integer` object that represents the specified `int` value. For example, we can create a new `Integer` object with value 5.

```
Integer number = new Integer(5);
```

`int`` intValue()` - Returns the value of this `Integer` as an `int`. For example, we can get the `int` value of the `Integer` object.

```
int intNumber = number.intValue();
```

Since version 1.5 of Java, the conversions between an `int` primitive and an `Integer` object can be handled by auto boxing and unboxing. When your code compiles, the code to convert back and forth between primitives and objects will automatically be created. If you are using earlier versions of Java or when writing code for the AP exam auto boxing and unboxing are not available so we would need to explicitly convert between primitives and objects using the methods given above. See examples on page 26 at the end of Section 2.4

One of the dangers of auto boxing and unboxing is that you don’t always realize the time penalty of converting between objects and primitives because the code to convert back and forth is hidden. For example, with auto boxing and unboxing both of the following sections of code are perfectly legal, however, the run time for the second section of code that uses an `Integer` object is 10 times slower than the first section of code that uses an `int` primitive.

```
int first = 5;
int second = first + 10;

Integer first = 5;
Integer second = first + 10;
```

Without auto boxing and unboxing, the second section of code would be:

```
Integer first = new Integer(5);  // or = Integer.valueOf(5);
Integer second = new Integer(first.intValue() + 10);
```

This code clearly shows the inefficiency of using `Integer` objects in this case. Generally, unless you are working with `ArrayLists` of numbers, you should use `int` primitives instead of `Integer` objects.

Here are some static methods and constants in the `Integer` class that you may find useful.

static int `parseInt(String str)` - Parses the string argument as a signed decimal integer. For example, to convert the String `"12345"` to an `int`

```
int number = Integer.parseInt("12345");
```

static int `parseInt(String str, `**int**` radix)` - Parses the string argument as a signed integer in the radix specified by the second argument. This method converts non base 10 numbers stored in a `String` to an `int`. For example, to convert a binary number in a `String` to a decimal `int` we could use the following.

```
int number = Integer.parseInt("101011", 2);
```

Note: Both of these `parseInt` methods will throw a `NumberFormatException` if the given `String` is not a valid number.

static `String toString(`**int**` num)` - Returns a `String` object representing the specified integer. For example:

```
String numbStr = Integer.toString(12345);
```

static `String toString(int num, int radix)`- Returns a string representation of the first argument in the radix specified by the second argument. This can be used to convert a number to a binary String (could also use `Integer.toBinaryString()`).

```
String binaryStr = Integer.toString(43, 2);
```

static int `MIN_VALUE` - A constant holding the minimum value of an `int (`-2^{31}).

static int `MAX_VALUE` - A constant holding the minimum value of an `int (`$2^{31}-1$).

Java also contains number classes for other integers including `Byte`, `Short` and `Long`. Each of these classes contains similar methods to the ones given above.

For real number objects, Java includes the `Float` and `Double` classes. Here are some of the details for the `Double` class. As you will see, the `Double` class is very similar to the `Integer` class with a few exceptions.

The `Double` class wraps a value of primitive type `double` in an object. It contains a single field whose type is `double`. In addition, this class provides several methods and some constants that can be useful when dealing with doubles. Here is a summary of some of these methods:

`Double(`**double**` value)` - Constructs a newly allocated `Double` object that represents the specified `double` value. For example, to create a new `Double` object with value 5.4:

```
Double number = new Double(5.4);
```

double `doubleValue()` - Returns the value of this `Double` as a `double`. For example, we can get the `double` value of the `Double` object.

```
double intNumber = number.doubleValue();
```

Just like `Integer` objects, since version 1.5 of Java, conversions between a `double` primitive and a `Double` object can be handled by auto boxing and unboxing. The same cautions mentioned earlier apply and generally, unless you are working with `ArrayLists` of numbers, you should use `double` primitives instead of `Double` objects.

Here are some static methods in the `Double` class that you may find useful.

`static int` `parseDouble(String str)` - Parses the string argument as a signed decimal integer. For example, to convert the String `"123.45"` to a `double`

```
double number = Double.parseInt("123.45");
```

Just like the `parseInt` methods in the `Integer` class, this method will throw a `NumberFormatException` if the given String is not a valid number.

`static` `String toString(`**`int`** `num)` - Returns a `String` object representing the specified double. For example:

```
String numbStr = Double.toString(123.45);
```

Here are some static constants in the `Double` class that you may find useful.

`Double.MIN_VALUE` - A constant holding the smallest positive non-zero value of type `double`, $2^{-1074} = 4.9 \times 10^{-324}$.

`Double.MAX_VALUE` - A constant holding the largest positive finite value of type `double`, $(2-2^{-52})\cdot 2^{1023} = 1.797\times 10^{308}$

`Double.NaN` - A constant holding a Not-a-Number (NaN) value of type `double`.

`Double.NEGATIVE_INFINITY` - A constant holding $-\infty$ of type `double`.

`Double.POSITIVE_INFINITY` - A constant holding $+\infty$ of type `double`.

5.6 The `BigInteger` Class

When we wrote a program to find the factorial of a number, we saw that if we were looking for numbers larger than `12!` the answer was larger than the largest available integer. If we used a `long` variable, we still found a problem when we tried to calculate `21!`. Although doubles can be used to store larger numbers, doubles only keep track of 15 significant digits so we need an alternative type of variable.

In other languages we need to write a lot of code to work with large integers and find values like `99!`. Fortunately in Java there is a built in object type called `BigInteger` that will handle large integers.

Here is an example of how to use `BigInteger` objects.

```
// Create two BigIntegers to add
BigInteger firstNumber = new BigInteger("999999999999999999999999");
BigInteger secondNumber = new BigInteger ("1");

// Add them and output the result
BigInteger total = firstNumber.add (secondNumber);
System.out.println ("The answer is: " + total);
```

will output `1000000000000000000000000`

Note: Since `BigInteger` is not part of the `java.lang` package, you will need to include the statement: **`import`** `java.math.BigInteger` at the top of your program when working with `BigIntegers`.

Since `firstNumber` and `secondNumber` are references to objects we have to use the `new` command to create the `BigInteger` objects. Even though `secondNumber` is small enough to be stored as a regular number, it was made a `BigInteger` to show that `BigIntegers` include small integers as well. Also notice the quotes around the initial value of the two numbers. Since the value to be stored in a `BigInteger` may be too large to be an integer constant, you must use String constants to represent the initial values of any `BigInteger`.

Furthermore, because `BigIntegers` are not primitives we can't use the traditional operators such as `+`, `-`, `*`, and `/`. Instead we use methods such as `add`, `subtract`, `multiply` and `divide`. Looking at the syntax of the `add` method:

```
     BigInteger total = firstNumber.add(secondNumber);
```

this means add the `secondNumber` to the `firstNumber` and then store the result in `total`. In languages such as C++ we can overload operators so that we can use the + operators etc. with new object types. In Java, this is partly allowed when working with String objects since we can use "+" to concatenate (“add”) two Strings.

Finally, we print the result using the statement:

```
System.out.println ("The answer is: " + total);
```

In this statement the `BigInteger total` will be converted to a `String` before it is concatenated to the string `"The answer is: "`. Like many object types the `BigInteger` class includes a method called `toString()` that converts a `BigInteger` to a `String`. Other `BigInteger` methods are given in Appendix F.

For large real numbers with unlimited precision, you can use the `BigDecimal` class. For more details on the `BigDecimal` class, see the Java API.

5.7 Questions, Exercises and Problems

1) Convert the following binary numbers to decimal:

a) `10111010.1101` b) `11001101.011`

2) Convert the following decimal numbers to binary:

a) `483.375` b) `78.8`

3) If you stored each of the numbers in question 2 in a **`double`** variable, which number would be stored more accurately? Clearly explain your answer

4 a) Given **`byte`** `number = -23;`, show the <u>memory bit pattern</u> for the variable `number`. Remember numbers are stored using two's complement.

b) Given **`short`** `newNumber = number;` where `number` is the variable given in part a), show the <u>memory bit pattern</u> for `newNumber`. Explain your answer.

5) Given the following bit patterns for `byte` variables, find the corresponding number.
a) `11111111` b) `01101101` c) `10100100`

6) Predict the exact output of the following code? Show any work and include a complete explanation of what is happening.

```
byte number = -125;
number -= 5;
System.out.println(number);
```

7) Predict the output of the following section of code. <u>Include an explanation of your output</u>.

```
for (int number = 1; number < 100; number++)
{
   float fraction = (float)1.0 / number;
   double total = 0;
   for (int count = 1; count <= number; count++)
       total += fraction;

   if (total == 1.0)
      System.out.println(number);
}
```

8) Given the memory bit pattern for the `float` variable `number` is `01000010101011101100000000000000`, what would be the decimal value of `number`? Show your work.

9) Given **`float`** `number = -2.7`, what would be the memory bit pattern for `number`? Show your work.

10) Given the memory bit pattern for the `float` variable `number` is `11111111100000000000000000000000`, what would be the decimal value of `number`? Show your work.

11) Prove that `number & -number` gives you the value of the right most bit of `number` by calculating the value of `number & -number` for the numbers 96, 72 and 63. To find the bit pattern for `-number` remember how negative numbers are stored using two's complement. Since these numbers are small, you only need to consider the right most 8 bits in your calculations.

12) Predict the output of the following:

```
int number = 137;
for (int shift = 1; shift <= 3; shift++)
{
   int left  = number << shift;
   int right  = number >> shift;
   System.out.printf ("%d: %4d %4d%n", shift, left , right);
}
```

Can you see any pattern? What happens to a number when you shift it 1 place to the left or 1 place to the right? Explain.

13) Assume the RGB colour for a single screen pixel is stored in an `int` variable called `pixel`. Working from left to right, the first 8 bits stores the alpha value (used for transparency), the next 8 bits stores the red value, the next 8 bits stores the green and the final 8 bits stores the blue value. For example, given the 32 bits we have:

AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB

a) Write the section of code to extract the 4 integer components (`alpha`, `red`, `green` and `blue`) from the single 32 bit `pixel` value.
b) Write the section of code that takes the 4 integer components (`alpha`, `red`, `green` and `blue`) and reconstructs them back into a single 32 bit `pixel` value.

14) Write the complete code for a method called `strToInt` that converts a number represented by a `String` in a given base into an integer. You should complete this method without using any of the built in Java methods.

You can assume two preconditions:

1) The given String will be a valid number for the given base.
2) The given base will be between 2 and 16 inclusive.

This method should have the same parameters and it should return the same result as `Integer.parseInt`. For example: `strToInt("1010", 2)` should return `14`.

15) Write the complete code for a method called `intToStr` that converts an `int` number with a given base into a `String`. You should complete this method without using any of the built in Java methods. You can assume that the given base will be between 2 and 10 inclusive. This method should have the same parameters and it should return the same result as `Integer.toString`.
For example: `intToStr (37, 2)` should return `"100101"`.

16) Write a program that gives the 32-bit pattern for a `float` variable in Java. Use the ideas presented in section 5.3.

17) Write a program that finds the decimal equivalent for a fraction. For example, given `3/4`, you should display `0.75`. Your program should handle repeating decimal values. For example, for `1/3` the answer should be `0.333...` To show the repeating decimal digits, you should enclose the repeating digits in parentheses. For example, for `1/3` you should display `0.(3)` and for `15/11` you should display `1.(36)`. Hint: Look for a repeating remainder when dealing with repeating decimal values.
Here are a few more examples:

Fraction	Decimal to display
39/7	5.(571428)
47/22	2.1(45)

Fraction	Decimal to display
56/7	8.0
461/26	17.7(307692)

18) If you were going to write your own `BigInteger` class, how would you represent each `BigInteger` in memory?

19) Write your own `add()` method for `BigInteger` using the results from question 18.

20) Use the `BigInteger` class to write a factorial method that can work with very large numbers.

21) Use the `BigInteger` class to find the largest 20-digit prime number.

Chapter 6 – Inheritance, Abstract Classes and Interfaces

One of the important features of Object Oriented Programming (OOP) is Inheritance which allows us to create new classes that inherit both data and methods from other classes. In this chapter we will learn more about inheritance and related topics including abstract classes and interfaces.

6.1 Inheritance

Instead of creating each new class from scratch, object oriented languages such as Java allow us to create subclasses that inherit both characteristics (variables) and behaviour (methods) from another class (superclass). Subclasses can also add new variables and methods. New methods with the same name and parameters will “override” methods in the superclass giving the new subclass specialized behaviour. Inheritance is an effective way of re-using or adding to both existing code and new code.

For example, in Grade 11 we created a `RobotPlus` class that extended the `Robot` class using the following heading:

```
public class RobotPlus extends Robot
```

In this case the “extends” means that the new `RobotPlus` class is a subclass of the `Robot` class and therefore inherits all of `Robot`’s data and methods. The data inherited from the `Robot` class included things such as a robot’s `name`, `colour` and `direction`. Inherited methods included `move`, `turnLeft`, `isWallAhead` and `pickUpItem`. The `RobotPlus` object could do everything that a `Robot` did.

We also added new data and methods to our `RobotPlus` class to give it additional features not available in the `Robot` class. In one example, we added a variable to keep track of the number of moves a robot made. In another example (`RobotTrack`) we keep track of the relative position of the robot on the grid by adding `changeNS` and `changeWE` variables. Methods we added to our `RobotPlus` class included `turnaround`, `turnToFace`, `goToCorner` and `pickUpAndMove`. In the `RobotTrack` class we added the `goToSpot` and `markThisSpot` methods. In order to keep track of the changes in position in the `RobotTrack` class, we also needed to “override” the `move` method so that it would update the new `changeNS` and `changeWE` variables every time it moved. To make a regular `Robot` move in this new `move` method we used **`super`**`.move()`to call the `Robot` class’ `move` method. You should look over sections 8.5 and 8.6 in the Grade 11 course pack to review some of these ideas.

We also used inheritance in the Connect Four program and the final project. To develop our own graphical applications we extended the built-in `JFrame` and `JPanel` classes. Since our new classes inherited both data and methods from the `JFrame` and `JPanel` classes, we easily created graphical windows with minimal code. We then added new data and methods to create specific behaviour for our projects. Depending on what you did for your final project, some of you may have extended other Java classes as well such as the `Rectangle` class.

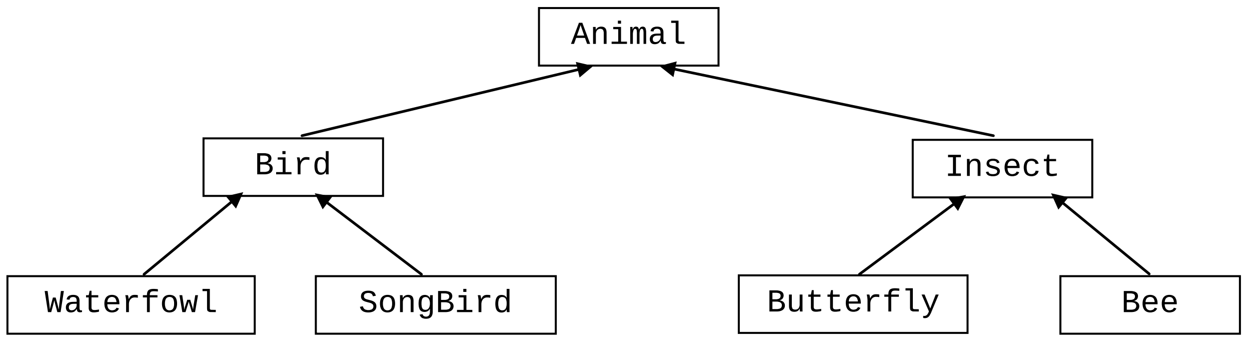
The big advantage of inheritance is that we don't need to duplicate code that already exists. The other advantage is that by creating a new subclass we leave the original superclass code alone. All of the new features are part of the subclass so if the new code has any bugs it will not affect the original code of the superclass.

In Java, a class can only extend one other class. What are some possible problems with letting a class extend more than one other class?

In the above examples we extended classes that already exist. In some cases you may want to take advantage of inheritance when designing your own code. By using inheritance you can minimize the amount of duplicate code. You can create superclasses with more general data and methods and subclasses with more specific data and methods. For example, in a chess game, you could create a `Piece` class that has the data and methods common to all chess pieces and then you could create subclasses such as `Pawn`, `Queen`, `Rook` etc. that have more specific methods (behaviour). To reduce the amount of duplicate code, we try to put any data and methods as high up in the class relationship as possible.

To determine when we can use inheritance we look for "is-a" relationships between classes. For example, a `Dog` class could be a subclass of a `Mammal` class because a "`Dog` is a `Mammal`". On the other hand a `Room` class would not be a subclass of `House` class since a `Room` is not a `House`. In this second case, the `House` and the `Room` classes have a "has-a" relationship which means the `House` class could contain a `Room` object as one of its instance variables. This "has-a" relationship is not inheritance.

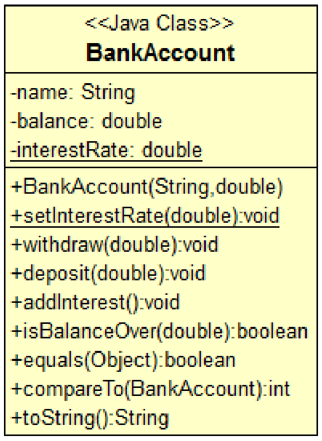
The inheritance relationship can have multiple levels with a subclass of one class being a superclass of another. These relationships can be shown in an inverted tree diagram with the superclasses on top. The "is-a" relationships between classes is valid up the tree. For example, a `SongBird` is both a `Bird` and an `Animal`.



The inheritance tree, or class hierarchy, can be as deep as needed. Methods and data/variables are inherited down through the levels. For example, the `Bird` class would inherit both data and methods from the `Animal` class and then the `SongBird` class would inherit data and methods from the `Bird` class. In this case, some of the data and methods inherited by the `SongBird` class would have been originally inherited by the `Bird` class from the `Animal` class. In general, the farther down in the hierarchy a class appears, the more specialized will be its behaviour.

6.2 UML Class Diagrams and a Shape Class Example

To show information about a class and inheritance relationships between classes we can use UML (Universal Modeling Language) class diagrams. In these diagrams, each class is divided into 3 sections. The top section identifies the class name and type of class (regular, abstract or interface). The middle section lists the attributes (data fields/variables) and the bottom section lists the operations (methods) including their parameter types. Here is an example showing the information for the `BankAccount` class developed in chapter 3. This diagram was created by Object Aid (an Eclipse plug-in) directly from Java code.



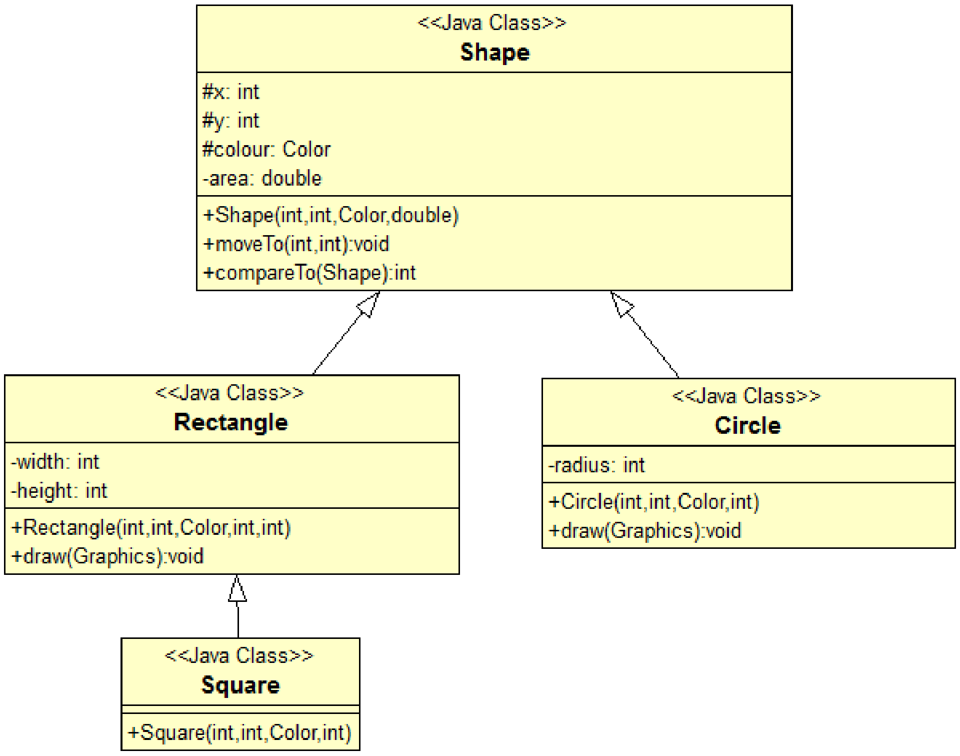
In this diagram, the symbols to the left of each variable or method indicate the visibility of each variable or method. The "+" is for `public` which is visible in every class, the "-" is for `private` which is only visible in its own class and the "#" is for `protected` which is visible in its own class and any subclasses. There are no protected variables in the `BankAccount` class but we will include protected variables in the next example. Also note that the underlined variables and methods are static.

With more than one class, superclasses are shown on top and subclasses are shown below. To show an inheritance relationship, an arrow is drawn from the subclass pointing to the superclass similar to the diagram given on the previous page. The inheritance arrow is a <u>solid line</u> with a <u>closed</u> arrow.

As an example, let's draw an UML class diagram with the following classes: `Shape`, `Square`, `Rectangle` and `Circle`. Think of the "is a" relationships to determine which classes are the superclasses and which classes are the subclasses. Once you have decided on the inheritance relationships, think about where you would put the following data fields: `colour`, `area`, `x`, `y`, `radius`, `height` and `width` and the following methods: `Shape`, `Square`, `Rectangle`, `Circle`, `draw`, `compareTo` and `moveTo`. Remember that data and methods in the superclass should be common to all subclasses. Data and methods in the subclasses are more specific.

The UML class diagram on the next page shows the relationships between the above classes. Try to sketch this diagram on your own before looking at the solution.

UML Class Diagram for the Shape classes



In this diagram all of the methods are `public` and most of the variables are `private`. Some of the variables in the `Shape` class are `protected` (#). Since a `protected` variable is visible in both the class where it is defined as well as any subclasses, the `x`, `y` and `colour` variables would be directly accessible in the `Shape` class as well as the `Rectangle`, `Circle` and `Square` subclasses. These variables will be used in the subclasses' `draw` methods when drawing each shape.

In this diagram, all shapes have the variables `x` and `y`, `area` and `colour` so these are defined in the superclass `Shape`. In addition to inheriting the four variables from the `Shape` class, `Rectangle` also has `width` and `height` variables and `Circle` has a `radius` variable. `Square` inherits the `width` and `height` variables from `Rectangle`.

The behaviour of `compareTo` is defined at the `Shape` level since `Shape` keeps track of the `area` that will be used for comparing. This will allow us to compare different types of `Shapes` by their areas. Although the `area` variable is kept in the Shape class, it will be determined in the `Rectangle` and `Circle` class constructors since we can't find a `Shape`'s area until we know what kind of shape it is. Similarly, the `draw` methods are defined in the `Rectangle` and `Circle` classes because you can't draw a shape until you know what kind of shape it is. Finally, since the `moveTo` method changes the values of the `Shape` variables `x` and `y`, it is also defined in the `Shape` class.

When you call a method for a certain type of object it first tries to use the version of this method defined inside its own class. This could be a new method or a method that overrides a method in the superclass. If the method is not defined in its own class, it uses the inherited method from its superclass. So if you ask a `Square` to draw itself it will use the `draw` method inherited from the `Rectangle` class.

Here is the actual Java code for the `Rectangle` class:

```
public class Rectangle extends Shape
{
    private int width;
    private int height;

    public Rectangle (int x, int y, Color colour, int width, int height)
    {
       super (x, y, colour, width * height);
       this.width = width;
       this.height = height;
    }

    public void draw (Graphics g)
    {
        g.setColor(colour);
        g.fillRect(x, y, width, height);
    }
}
```

The class heading shows that the `Rectangle` class is an extension of the `Shape` class, inheriting both data and methods from its superclass. In addition to its inherited variables, `Rectangle` defines two new private instance variables (`width` and `height`). The first line of the constructor is used to initialize variables defined in the superclass:

```
super (x, y, colour, width * height);
```

The `super` key word calls the constructor of the superclass `Shape` passing the `x`, `y`, `colour` and calculated area to initialize the inherited variables. Since `area` is determined by the type of shape, the `Circle` constructor will be different. For example, this would be the first line of the `Circle` constructor

```
super (x, y, colour, Math.PI * radius * radius);
```

The next two lines of the constructor initialize the variables defined in the `Rectangle` class. The “`this`” modifier is used to clarify that we are referring to this object’s instance variables. In many cases, the “`this`” modifier is optional but here it is needed because the parameter variables have the same names as the instance variables.

Finally, the `draw` method draws a rectangle in the given colour. Since the `x`, `y` and `colour` variables were defined in the `Shape` class; we needed to make them `protected` in order to access these variables directly in the `Rectangle` class.

The `Rectangle` class will also inherit the `moveTo` and `compareTo` methods from the `Shape` class. The `moveTo` method changes the position of a `Rectangle` and the `compareTo` method compares the area of this `Rectangle` to the area of any `Shape` object. This allows us to compare this `Rectangle` to another `Rectangle` or even a `Circle` or a `Square`. See the example on the next page.

Here is some sample code using these classes. The following code will draw 3 different shapes. Describe the position, colour and shape of each of these shapes. You can assume that g is a valid `Graphics` object.

```
Rectangle myRect = new Rectangle(0, 0, Color.BLUE, 100, 210);
Circle myCirc = new Circle(200, 200, Color.GREEN, 81);
Square mySquare = new Square(200, 400, Color.RED, 145);
if (myRect.compareTo(myCircle) >= 0 && myRect.compareTo(mySquare) >= 0)
    myRect.moveTo(400,400);
else if (myCirc.compareTo(mySquare) >= 0)
    myCirc.moveTo(400,400);
else
    mySquare.moveTo(400,400);
myRect.draw(g);
myCirc.draw(g);
mySquare.draw(g);
```

Object References with Inheritance

The type of an object reference doesn't always have to be the same as the type of the object that it refers to. For example, since a `Circle` is a `Shape`, we can use a `Shape` reference to refer to a `Circle` object. For example:

```
Shape myShape = new Circle(10,10,Color.RED,30)
```

This code is perfectly legal. However, in this case `myShape` can only have `Shape` behaviour so:

`myShape.moveTo(45,90)` would work but
`myShape.draw(g)` would not.

However, if we knew `myShape` referred to a `Circle` we could cast it to `Circle` before calling the `draw` method. Therefore,

`((Circle)myShape).draw(g)` would work.

To avoid casting we usually declare an object reference as the same type as the object that it refers to. However, we will see in section 6.3, that we can use abstract classes and abstract methods in order to declare `myShape` as a `Shape` reference and then call `myShape.draw(g)` without having to cast `myShape` to `Circle`.

Also, when working with `ArrayLists`, we often use the following:

```
List<String> myList = new ArrayList<String>();
```

Since most of the `List` behaviour is the same as the `ArrayList` behaviour, this will usually not cause a problem. The `List<String> myList` code is often used for a parameter in a method heading so that the method can handle any type of `List` (e.g. `ArrayList` or `LinkedList`).

6.3 Abstract Classes

In a class hierarchy, if you only need to create instances of subclasses, you can make the superclass an abstract class. For example, in our earlier example, the `Shape` class could have been an abstract class since `Shape` is a general class that we won't be creating instances of. Instead we create instances of the subclasses such as `Rectangle` and `Circle`.

Abstract classes can contain variables, regular methods and abstract methods. For an abstract method only the heading is given with the return type, method name and any parameters. No body or code is given at this stage since the specific behaviour of an abstract method will be defined by an overriding method in a non-abstract subclass. By including an abstract method heading in an abstract class we are ensuring that any non-abstract subclass will define the behaviour of this method.

If a class extends an abstract class it will inherit the abstract class' data and non-abstract methods. Any non-abstract subclasses of an abstract class must override any abstract methods since the abstract method's behaviour has not been defined yet.

This can be illustrated with our shapes example. If we made the `Shape` class `abstract` we would put an abstract `draw` method in this class. Since we can't draw a `Shape` until we know what kind of shape it is, the `draw` method is a good example of an abstract method. Including this method in the superclass forces all of the subclasses to include a `draw` method. So, if we had an `ArrayList` of `Shape` objects we would be able to draw all of the shapes regardless of what type of `Shape` they were. For example, given the `Shape` class is defined as abstract with an abstract `draw` method.

```
public abstract class Shape implements Comparable<Shape>
{
    // Other code not shown

    public abstract void draw (Graphics g);
}
```

would force the `Rectangle` and `Circle` classes to include a `draw` method so the following code would work:

```
    List<Shape> allShapes = new ArrayList<Shape>();

    allShapes.add(new Rectangle(200, 300, Color.BLUE, 100, 300));
    allShapes.add(new Circle(100, 200, Color.RED, 75));
    allShapes.add(new Square(200, 100, Color.GREEN, 150));

    for (Shape nextShape : allShapes)
    {
        nextShape.draw(g);
    }
```

Even though `nextShape` is a `Shape` reference, `nextShape.draw(g)`will call the appropriate `draw` method depending on what type of object `nextShape` refers to.

Since we are using a `Shape` reference to refer to each element in the `ArrayList` of `Shapes`, the compiler looks for a `draw` method in the `Shape` class. Without the abstract method in the `Shape` class, the above code would not work, even if we knew that `Rectangle` and `Circle` contained a `draw` method. Abstract methods tell the compiler that this method will be included in the more specific non-abstract subclasses.

In this example, the `allShapes ArrayList` contains references to `Shape` objects. Since `Rectangle`, `Circle` and `Square` objects are all `Shape` objects and because a superclass reference type can refer to a subclass object, we can add a mixture of these objects to our `List`.

Since an abstract class may contain abstract methods with undefined behaviour we cannot create instances of an abstract class. So the following code would not work:

```
Shape myShape = new Shape(200, 100, Color.CYAN, 0);
```

Instead we extend the abstract class and create instances of the subclasses which will have all of their behaviour defined since non-abstract classes must override any abstract methods from their superclass.

Here is another example of how we can use abstract classes and abstract methods. Given the following declarations:

```
public abstract class Vehicle
public class Bicycle extends Vehicle
public class Car extends Vehicle
```

then we can create a `Vehicle` class reference that can refer to either a `Bicycle` or a `Car`. So both of the statements given below would be allowed:

```
 Vehicle myRide = new Bicycle();
 Vehicle myRide = new Car();
```

Then, if we include an abstract `goToWork` method in the `Vehicle` class, we know that the non-abstract sub-classes `Bicycle` and `Car` must override the `goToWork` method. This allows us to wait until run time to decide what kind of object that we want to use. For example, in the following code:

```
Vehicle myRide;
if (isRaining)
   myRide = new Car();
else
   myRide = new Bicycle();

myRide.goToWork();
```

When it is raining, `myRide` will be a `Car` and if it is not raining `myRide` will be a `Bicycle`. The object that we end up using and the resulting behaviour depend on whether it was raining or not. This is an example of **Polymorphism**. Using polymorphism can make your programs more flexible.

6.4 Interfaces

In Java an interface is like a pure abstract class. Since it is purely abstract, it has no defined data (instance variables) or behaviour (non-abstract methods). It can only contain static constants and abstract methods. By default all interface methods are abstract so the abstract modifier is not needed for methods.

If a class was allowed to extend more than one class, we saw earlier that this could cause a conflict if the two superclasses being extended shared the same names for either their data or methods. In this case the subclass wouldn't know which version of the data or method to inherit. However, since there is no data or implemented methods in an interface, a class can implement more than one interface.

To agree to take on the behaviours of an interface, we use the key word "`implements`". If a class wants to implement more than one interface, each interface is listed, separated by a comma. For example:

```
public class Time implements Comparable<Time>, Serializable
```

Just like a non-abstract class must implement any abstract methods of its superclass, a class that implements an interface must implement all methods of the interface. For example, when we created a class that implemented `Comparable` we had to include a `compareTo` method.

Interfaces can be used to indicate that a class will include certain behaviors. For example, methods like `Collections.sort` and `Collections.binarySearch` will only work with `Lists` of `Comparable` objects since these objects are guaranteed to have a `compareTo` method that can be used to determine the order of the `List`.

Another built in interface mentioned above is `Serializable`. A `Serializable` object is an object that can be written to or read from a file. If the data types of the instance variables of your class are primitives or `Serializable` objects themselves, such as `Strings`, no further action is required. In this case you are just confirming that your object can be written to a file. Classes that require special handling during the serialization and deserialization process must implement the `writeObject`, `readObject` and `readObjectNoData` methods. However, in most cases, these methods are not necessary. For more information on reading and writing objects to a file and the `Serializable` interface, see Appendix E.4.

When working with Graphical User Interfaces (GUI's) commonly used interfaces include `MouseListener` or `MouseMotionListener`. If you want your `JPanel` to respond to mouse events such as `mousePressed` or `mouseReleased` you can implement a `MouseListener`. If you want to handle events such as `mouseMoved` and `mouseDragged` you can implement `MouseMotionListener`. In both cases, if you implement the given interface, you are agreeing to implement methods to handle all of the related mouse events.

Unfortunately for the `MouseListener` there are a total of 5 methods that you need to override. The other 3 methods you need to include are `mouseClicked`, `mouseEntered` and `mouseExited`. Even if you aren't going to use these methods you still need to include a heading and an empty body since you agreed to include all of the `MouseListener` methods when you implemented the interface. To minimize the number of "dummy" methods in your code you can use a `MouseAdapter` instead that includes some of these extra dummy methods for you. See the Java API for more details on using `MouseAdapters`.

In addition to using the built-in interfaces, we can also write our own interfaces. See section 6.5 for an example of creating your own interface.

When creating a hierarchy of interfaces we may want an interface to extend another interface. This is allowed. For example:

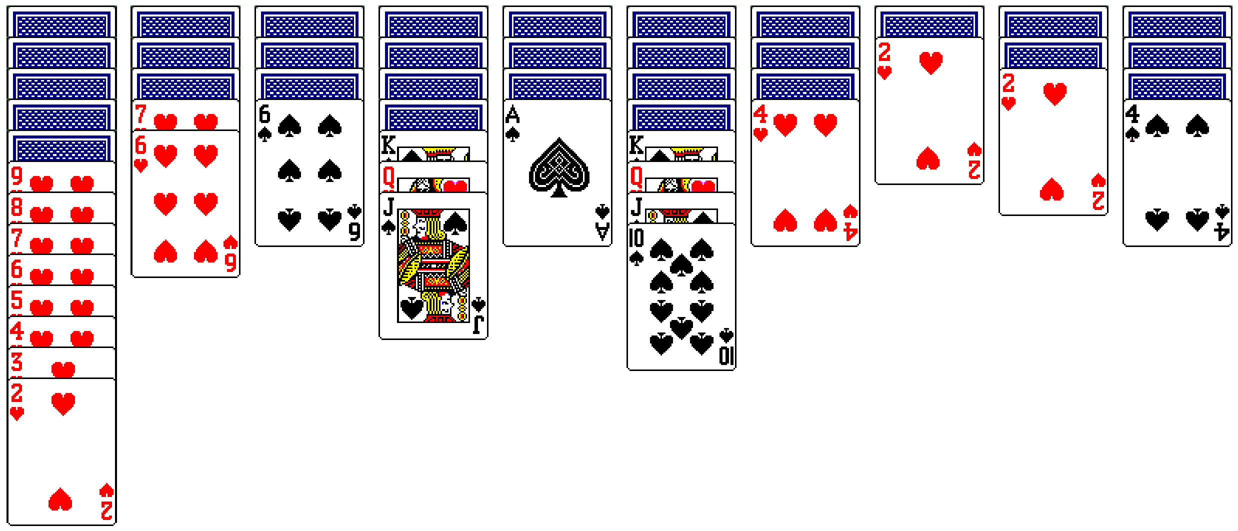
```
public interface A extends B
```

Since the Interface A is abstract it can't implement any methods including any abstract methods in B. Therefore, a non-abstract class that implements A would need to override all of the methods from both A and B.

6.5 Creating your own Interfaces

In addition to using the built-in interfaces such as `Comparable`, `Serializable` and `MouseListener`, we can also write our own interfaces.

In card solitaire games such as Klondike and Spider Solitaire, the user is allowed to move individual cards or whole groups of cards from one section to another. We could use a `Movable` interface to describe the required behaviour of these "movable" objects. For example, here is a sample Spider Solitaire game in progress.



In this game we use a `Stack` object to keep track of each column of `Cards`. Since a `Stack` object is similar to a `Hand` object which is just a list of `Cards`, `Stack` will extend the `Hand` class. In the game of Spider Solitaire we can always pick up the top card from a `Stack`. For example, in the above layout we can pick up the 2H, 6H, 6S, JS, etc. We can also pick up descending runs of `Cards` in the same suit such as the 9H to 2H run in the first column or the 7H 6H run and the JS TS run in the 2nd and 6th columns respectively. To deal with picking up either a `Card` or another sub `Stack` from a `Stack`, the `Stack` class includes a `pickUp` method that returns a `Movable` object. This will work if both the `Card` and `Stack` classes implement `Movable`.

Once this `Movable` object is picked up, we need to be able to drag it and drop it onto its new location. Also, before dropping the object we need to check if the drop location is valid. Finally, to show the object as it moves, we need to be able to draw that object. Here is a sample `Movable` interface that includes all of these behaviours.

```
public interface Movable
{
   public void move(Point initialPos, Point finalPos);
   public void draw(Graphics g);
   public boolean intersects(Stack hand);
   public boolean canPlaceOnStack(Stack hand);
   public void placeOnStack(Stack hand);
}
```

The `move` and `draw` methods will be used when dragging the `Card` or the `Stack`, the `intersects` and `canPlaceOnStack` methods will be used to check if each moving object can be dropped and the `placeOnStack` method will be used to place the object in its final position. Since both the `Card` and `Stack` classes implement `Movable`, they will need to include all of these methods. This allows us to simplify our code by only writing code for a `Movable` object instead of writing separate code to handle both a moving `Card` and a moving `Stack`. In our main program, we would create a `Movable` object for the selected item that we are picking up:

```
          private Movable selectedItem;
```

When we click on a `Stack` in the `mousePressed` method, we use its `pickUp` method to return the `selectedItem`.

```
          selectedItem = selectedStack.pickUp(selectedPoint);
```

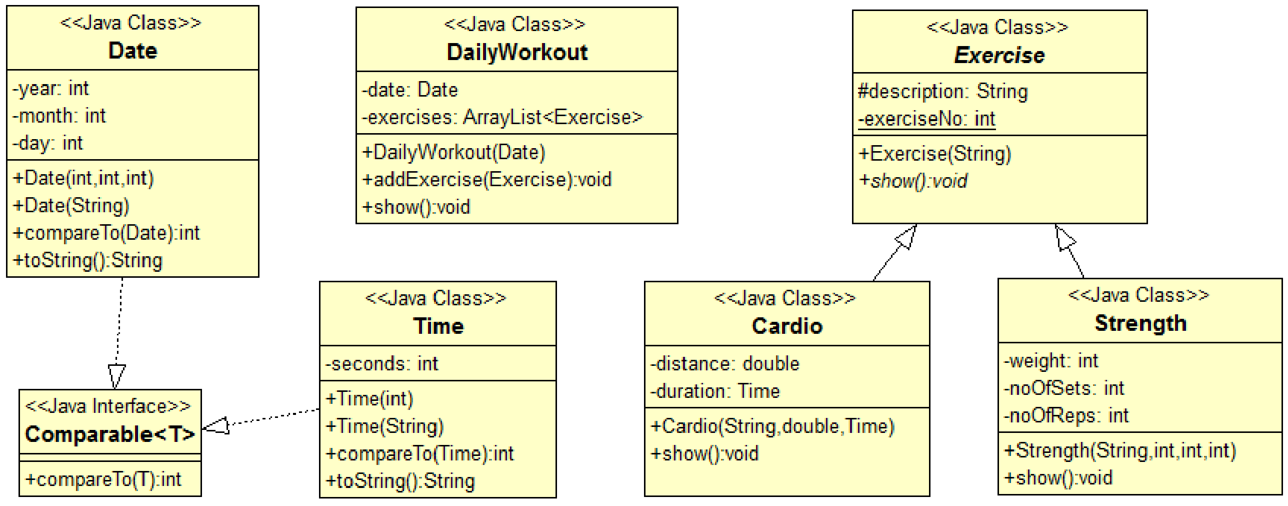
The `pickUp` method needs the `selectedPoint` parameter to know whether we are selecting the top `Card` or a sub `Stack` of `Cards`. This method would actually return a `Card` or a sub `Stack` but the main program would just treat it as a `Movable` object. It wouldn't care which specific object it was since it would treat all `Movable` objects in the same way using the methods specified above in the `Movable` interface.

6.6 `Exercise` Class Example

Assume that we want to create a Java program to keep track of a person's exercise routine. Here are 7 different classes that could be used to do this.

`WorkoutDiary` – keeps track of all of your workouts including a summary of each type of exercise.
`DailyWorkOut` – keeps track of each daily workout. Daily workouts are made up of one or more exercises.
`Exercise` – a general exercise class
`Strength` – a more specific exercise class to keep track of strength exercises such as bench presses, curls and push ups.
`Cardio` – a more specific exercise class to keep track of cardio exercise such as running, biking and swimming.
`Date` – keeps track of a date that will be used by the `DailyWorkOut` class.
`Time` – keeps track of a time that will be used by the `Cardio` class.

Here is a UML diagram to show the relationships between these classes. To keep things simple the `WorkoutDiary` class has not been included in this diagram. The main purpose of these classes is to present a simple example that demonstrates some of the concepts presented in this chapter so some items may be missing.



Looking at this diagram you can see some "is a" relationships since both `Cardio` and `Strength` are subclasses of `Exercise`. There are also a few aggregation or "has a" relationships. `Cardio` has a `Time` object and `DailyWorkout` has both a `Date` object and an `ArrayList` of `Exercise` objects. These aggregation relationships can be shown by an arrow with a <u>diamond</u> arrow head but they are not shown on this diagram. Since both the `Time` and `Date` classes implement `Comparable` these relationships to the `Comparable` interface are shown with <u>dotted</u> lines and <u>closed</u> arrow heads.

Recall, the symbols beside each variable and method indicate their visibility. Also, static variables and methods are underlined. Finally, you may have noticed that abstract classes (`Exercise`) and methods (`show`) are shown in *italics*.

Below is the code for the `DailyWorkout`, `Exercise`, `Cardio` and `Strength` classes. To save space, class and method comments have been omitted.

```java
public class DailyWorkout
{
  private Date date;
  private ArrayList<Exercise> exercises;

  public DailyWorkout(Date date)
  {
     this.date = date;
     exercises =
            new ArrayList<Exercise>();
  }

  public void addExercise(Exercise
                           exercise)
  {
     exercises.add(exercise);
  }

  public void show()
  {
     System.out.println
          ("Workout for: " +  date);
     for (Exercise next : exercises)
        next.show();
  }
}

public abstract class Exercise
{
  protected String description;
  private static int exerciseNo = 0;

  public Exercise (String description)
  {
     exerciseNo++;
     this.description = exerciseNo +
           ") " + description;
  }

  public abstract void show();

}
```

```java
public class Cardio extends Exercise
{
  private double distance;
  private Time duration;

  public Cardio(String description,
       double distance, Time duration)
  {
     super(description);
     this.distance = distance;
     this.duration = duration;
  }

  public void show()
  {
      System.out.printf("%s: %.1fkm in %s%n",
          description, distance, duration);
  }
}

public class Strength extends Exercise
{
  private int weight;
  private int noOfSets;
  private int noOfReps;

  public Strength(String description,
      int weight, int noOfSets, int noOfReps)
  {
      super(description);
      this.weight = weight;
      this.noOfSets = noOfSets;
      this.noOfReps = noOfReps;
  }

  public void show()
  {
      System.out.printf("%s: %dkg %d sets
          of %d%n", description, weight,
                   noOfSets, noOfReps);
  }
}
```

For the `Time` and `Date` classes see the variables and methods on the UML diagram. The `Time` class would be similar to the `Time` class given in section 6.4 of the Grade 11 course pack. For the `toString` method in the `Date` class, you can assume that the return value would use a 3 digit month (e.g. Nov 5, 2014).

Looking at the above code, in the `DailyWorkout` class we have variables to keep track of the date and a list of exercises that make up the workout. The `exercises` `ArrayList` is a list of `Exercise` objects which can include a combination of both `Cardio` and `Strength` objects. When we display the `exercises` in the `show` method, the code will know which version of the `show` method to call for each `Exercise` depending on whether it is a `Cardio` or `Strength` object.

The Exercise class is an abstract class that keeps track of the description of each exercise. Since description is common to all exercises, it is placed in the Exercise class. The visibility is declared as protected so that this variable can be directly accessed in the Cardio and Strength subclasses. This class also contains a static variable called exerciseNo that keeps track of the next exercise number. Since it's static, there will be only one copy of this variable that will be shared by all instances of the Cardio and Strength classes. The exerciseNo is originally set to 0 and it is updated every time an Exercise object is created. If you wanted to re-start the numbers for each workout, you could add a static method to reset the exerciseNo back to 0.

Finally, the Exercise class contains an abstract method called show. Any subclass of Exercise such as Cardio and Strength must override this method. This allows us to declare the exercises list in DailyWorkout as an ArrayList of Exercise objects which could contain both Cardio and Strength objects.

The Cardio class contains instance variables to keep track of the distance ran, cycled or swam and the time it took to complete this exercise. The Strength class contains instance variables to keep track of the weight lifted, the number of sets and the number of repetitions for each set. Both classes include constructors that set their description variables by calling the Exercise constructor using the key word super.

Finally they both include a show method that displays important information about each exercise. Each show method can access the description variable from the superclass since this variable was declared as protected.

Look over this code carefully to make sure that it makes sense to you. Question 14 in the questions at the end of the chapter refers back to this code.

Can you think of any other classes that you may want to include in an exercise program? Are there any other variables or methods that you may want to add to any of the given classes?

6.7 Object Oriented Design

The following is a brief discussion of some techniques that can be used when designing programs with objects.

One simple idea to help you figure out which objects to include in your code as well as what variables and methods to include in each object, is to make a list of all words associated with your project. You could brainstorm with members of your group to try to come up with as many words as possible.

Your next step is to put these words into two lists, one with nouns and one with verbs. Don't worry about words that don't fit into either list. Also, you may find that some words can fit into both lists.

Using the noun list, your next step is to identify any potential objects and variables. Using the "has a" relationship, you can see which nouns are best suited to be objects and which nouns would be part of an object or a variable. Nouns with multiple occurrences in your project such as students or products could be potential objects. You may even find that some objects will be part of another object. Once you have figured out which objects you want to include, you can look for any "is a" relationships to identify any inheritance relationships. If you find two or more objects share the same variables you may find you can create a superclass that contains these common variables and subclasses with more specific variables.

You final step is to use the list of verbs to come up with potential methods for your objects. If you have any inheritance relationships remember to try to put methods as high up in the inheritance tree as possible. CRC cards (see below) can help to identify which objects are responsible for which tasks/methods.

Another technique that is helpful when designing programs with objects is to create CRC (Classes, Responsibilities and Collaborators) cards for each object. You start by creating an index card for each class. Next, looking at the tasks that you need to perform to complete your project (use the verbs given earlier) you identify which class is responsible for this task and then write this responsibility (task) on the appropriate class' index card. For each responsibility listed you should also identify which other classes (collaborators) are needed to complete this task. The collaborators needed for each responsibility (task) should be listed beside each responsibility.

When you identify a collaborator for a responsibility you then get the index card for the collaborator and add the task it would need to complete when collaborating as one of its responsibilities. This process continues until you have found out who is responsible for all of the required tasks and you have dealt with all of the collaborators.

For example, assume you are creating a Poker game and you have `Card`, `Deck`, and `PokerHand` classes. One of the tasks that you need to complete is to evaluate a hand to see what kind of poker hand it is (e.g. Flush, Two of a Kind, Straight etc.). Who would be responsible for this task? Since it would be the `PokerHand` class, you would list this task (e.g. `getType`) as one of `PokerHand`'s responsibilities. Now, to complete this task, you would need to know information about each `Card` in the hand so you would list the `Card` class as a collaborator beside the `getType` task. Then, in order for the `Card` class to collaborate properly you would need to know the card's rank and suit so you would add a `getRank` and a `getSuit` method to the list of responsibilities on the `Card` class' index card. Since no collaborators would be needed for these two new responsibilities we are finished with the task.

You would continue this process for all of the identified tasks. For example, another task is to add a `Card` to a `PokerHand`. Which class would be responsible for this task and which class(es) would be the collaborators. For any collaborators, are there any new responsibilities to add?

In question 15 at the end of the chapter, you are given a list of words for a Blackjack game and you are asked to determine which objects, variables and methods you would need for this game.

6.8 Questions, Exercises and Problems

1) For each of the following indicate whether the "extends" relationship is valid (V) or not valid (NV).

Banana **extends** Fruit	_____	Baseball **extends** Sport	_____
Tool **extends** Saw	_____	Whale **extends** Fish	_____
Mall **extends** ToyStore	_____	Spaghetti **extends** Pasta	_____

2) For each of the following lists of classes, draw a simple inheritance hierarchy diagram (no data or methods required) to show the relationships between the classes. In some cases there will be no connection between the classes. **Hint:** Use the "is a" relationship to identify subclass and superclass relationships.

a) `Student`, `Teacher`, `Person`, `Educator`, `Principal`, `PartTimeStudent`, `FullTimeStudent`

b) `Fish`, `Feline`, `Lion`, `Cardinal`, `Mammal`, `Salmon`, `Tiger`, `Man`, `Halibut`, `Animal`, `Bird`, `Dolphin`, `Robin`, `Jaguar`, `Trout`

c) `House`, `Kitchen`, `Building`, `Room`, `HighSchool`, `Classroom`, `School`, `Desk`, `Bathroom`, `Neighbourhood`, `ElementarySchool`

3) Given the classes Animal, Bird and Amphibian, the data variables name, weight, noOfLegs and wingspan, and the methods Animal, Bird, Amphibian, fly, grow, and metamorphosize, draw a UML class diagram with both data and methods shown. Parameters for methods are not required.

4) Some object oriented languages such as C++ allow multiple inheritance where a class can extend more than one superclass. In Java, only single inheritance is allowed and a class can only extend one other class. What are some possible problems with multiple inheritance?

5) The `final` modifier used for constants can also be used for both classes and methods. For example:

```
public final class FirstClass or

public final void myMethod()
```

What do you think it means if we make a class final?
What does it mean if we make a method final?

6) Can a subclass access the private data of its superclass? Can you give a subclass direct access to data in its superclass without making the data public? Clearly explain your answer.

7) If a subclass overrides a method from its superclass, how can you call the superclass' version of this method inside code in the subclass?

8) Given the following class code:

```
public class Base
{
    private int value;

    public Base (int value)
    {
        this.value = value;
    }

    public void methodOne ()
    {
        System.out.println ("A " + value);
    }


    public void methodTwo ()
    {
        System.out.println ("B");
    }


    public void methodThree ()
    {
        System.out.println ("C");
    }
}
```

```
public class First extends Base
{
    private int amount;

    public First(int data)
    {
       super(data + 1);
       amount = data;
    }
    public void methodOne ()
    {
        super.methodOne ();
        System.out.println ("M" + amount);
    }

    public void methodTwo ()
    {
        System.out.println ("N");
        methodThree ();
    }
}

public class Second extends First
{
    public Second()
    {
        super(10);
    }
    public void methodThree ()
    {
        System.out.println ("X");
    }
}
```

Show memory diagrams for the following code:

```
    Base one = new Base (5);


    Base two = new First (7);


    Base three = new Second ();
```

Predict the output of the following sections of code:

	Output		Output
`one.methodOne ();`		`one.methodTwo ();`	
`two.methodOne ();`		`two.methodTwo ();`	
`three.methodOne ();`		`three.methodTwo ();`	

	Output
`one.methodThree ();`	
`two.methodThree ();`	
`three.methodThree ();`	

9) Given the following code for a `Pet`, `Fish`, `Dog` and `Snake` class:

```java
public abstract class Pet
{
   private String name;

   public Pet(String name)
   {
      this.name = name;
   }

   public String getActivity()
   {
      return "Do nothing";
   }

   public void feed()
   {
      System.out.println("Feed me " +
              getFood());
   }

   public String toString()
   {
      return "My name is " + name +
        " and I like to " + getActivity();
   }

   public abstract String getFood();

   public abstract void speak();
}
```

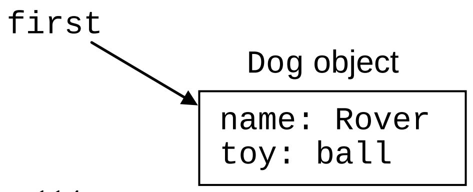
```java
public class Dog extends Pet
{
   private String toy;

   public Dog(String name, String toy)
   {
      super(name);
      this.toy = toy;
   }

   public Dog(String name)
   {
      this(name, "ball");
   }

   public String getActivity()
   {
      return "Play with my " + toy;
   }

   public String getFood()
   {
      return "Dog biscuits";
   }

   public void speak()
   {
      System.out.println("Woof woof");
   }
}
```

```java
public class Fish extends Pet
{
   public Fish(String name)
   {
      super(name);
   }

   public String getActivity()
   {
      return "Swim";
   }

   public String getFood()
   {
      return "Fish food";
   }

   public void speak()
   {
      System.out.println("Glub Glub");
   }
}
```

```java
public class Snake extends Pet
{
   public Snake(String name)
   {
      super(name);
   }

   public String getFood()
   {
      return "Mice";
   }

   public void speak()
   {
      System.out.println("Sssssss");
   }
}
```

a) Using the code given above, draw a memory diagram for each of the following Pet objects. The first one is completed to help get you started.

```java
// Create some Pet objects
Pet first = new Dog("Rover");
```



9a) continued

```
Pet second = new Snake("Slither");
```

```
Pet third = new Fish("Wanda");
```

```
Pet fourth = new Dog("Spot", "chew toy");
```

b) Predict the output of the following code:

```
 // Create an ArrayList of Pets and add some pets in order
ArrayList<Pet> pets = new ArrayList<Pet>();
pets.add(first);           pets.add(second);
pets.add(third);           pets.add(fourth);

// Let each Pet speak
for (Pet nextPet : pets)
   nextPet.speak();
```

```
// Feed each pet
for (Pet nextPet : pets)
   nextPet.feed();
```

```
// Display each pet's information
for (Pet nextPet : pets)
   System.out.println(nextPet);
```

c) Predict the output of the following code:

```
Pet myPet = new Dog("Fido");
if (Math.random() < 1.0/3)
    myPet = new Snake("Greg");
else if (Math.random() < 0.5)
    myPet = new Fish("Martha");
myPet.speak();
```

10) Given the following class code:

```
public abstract class First
{
    public void methodOne ()
    {
        System.out.println ("1");
    }

    public void methodTwo ()
    {
        System.out.println ("2");
    }

    public abstract void methodThree ();

    public abstract void methodFour ();
}

public class Second extends First
{
    public void methodOne ()
    {
        System.out.println ("3");
    }

    public void methodThree ()
    {
        System.out.println ("4");
        methodFour ();
    }

    public void methodFour ()
    {
        System.out.println ("5");
    }
}
```

```
public class Third extends Second
{
    public void methodTwo ()
    {
        System.out.println ("6");
    }

    public void methodFour ()
    {
        System.out.println ("7");
    }
}

public class Fourth extends First
{
    public void methodOne ()
    {
        System.out.println ("8");
    }

    public void methodFour ()
    {
        System.out.println ("9");
    }
}
```

a) Explain any errors in the above code. Even though no constructors are shown, this is not an error since a default constructor with no parameters will be created for each class automatically.

b) Assuming the code with errors was removed from the above code; predict the output of the following sections of code. Each section of code is separate. If there is no output, explain why! Be <u>specific</u> in your explanations.

<u>Output</u>

```
a) First one = new First ();
   one.methodOne ();


b) First two = new Second ();
   two.methodOne ();
   two.methodTwo ();
   two.methodThree ();


c) First three = new Third ();
   three.methodOne ();
   three.methodTwo ();
   three.methodThree ();
```

11) Given the following class and interface code:

```
public abstract class Vehicle
{
    public abstract void ride ();
}

public abstract class Animal
{
    public abstract void eat ();
    public abstract void reproduce ();
}

public abstract class GardenAppliance
{
    public void putInShed ()
    {
        // Code to put in shed
    }
}
```

```
public interface Floatable
{
    public void floatOnWater ();
}

public interface Fuelable
{
    public void addFuel ();
}

public interface Flyable
{
    public void fly ();
}
```

a) Complete the following class headings:

```
public abstract class Boat


public class MotorBoat


public class Canoe


public class Car


public class Airplane


public abstract class Bird


public class Duck


public class GasLawnMower


public class ElectricEdgeTrimmer
```

b) Given the above headings, what methods would you <u>need</u> to include in each class given above. Put your answer underneath each of the above classes.

12) Given the following classes and interfaces, draw a UML class diagram showing the class and interface relationships. Variables and methods are not required. You should label the abstract classes and interfaces accordingly. Don't forget to use the proper types of arrows. (solid for inheritance and dotted for interfaces)

`Walkable, Petable, Pet, PetBird, Parrot, PetFish, Dog, Poodle, PetFood, GoldFish, ClownFish, Bulldog and PetOwner`

13) What are the similarities and differences between abstract classes and interfaces?

14) Referring to the `DailyWorkout`, `Exercise`, `Cardio`, `Strength`, `Date` and `Time` classes covered in section 6.6 and given the following main program code:

```
Date date = new Date("2014/11/16");
DailyWorkout first = new DailyWorkout(date);
first.addExercise(new Strength("Bench Press", 80, 2, 12));
first.addExercise(new Cardio("Run", 7.25, new Time("43:12")));
first.addExercise(new Strength("Curl", 35, 3, 15));
first.show();
```

a) Draw a neat and labeled memory diagram to show a memory trace for the main program code given above. **Note**: You can show any `String` objects like primitive variables but all other objects should be shown properly with each object type labeled. Remember, each "**new**" in the above code and in the class code should create a new object. Assume `toString` returns "mmm dd, yyyy" for `Dates` and "mm:ss" for `Times`.

b) In your diagram in 14a, clearly label at least one example of the following:
- i) An instance of the `Cardio` class
- ii) A reference to a `Date` object
- iii) An instance variable
- iv) A class variable

c) Predict the exact output of the main program code given above.

d) For each of the following separate modifications to the code in Section 6.6, clearly explain what would be the resulting error (be specific) and why this error would occur.
- i) If you removed the heading: **`public abstract void`** `show()` from the `Exercise` class.
- ii) If you changed the modifier on the `description` variable from **`protected`** to **`private`** in the `Exercise` class.
- iii) If you removed the `compareTo()` method from the `Time` class.

15) Assume you want to write a simple Blackjack game. Based on your initial brainstorming you came up with the following nouns and verbs relating to Blackjack:
deal, bust, card, hand, deck, hand value, number of decks, shuffle, suit, rank, dealer, player, black jack hand, table, ace, twenty-one, face card, blackjack, cards in your hand, cards in the deck, number of cards left in the deck, hold, take a card, flip a card, face down, face up, win, lose, show hand, busted, and top (next) card in deck.

a) Based on the nouns from the above list, identify the classes that you think should be included in a Blackjack game.

b) For 5 of these classes (pick the 5 you think are most important), identify the data fields that you should include in each class. Use some of the nouns given above to help think of the required data fields.

c) For the 5 classes identified in part b), identify the methods (behaviour) that you should include in each class. Use some of the verbs given above to help think of the required methods. If it helps, create a CRC card for each class in part b).

Appendix A -- Java Style Guide

A.1 Naming Identifiers

Identifiers are used to name classes, variables, constants and methods in your Java programs. When picking identifiers, you must follow the following guidelines. Identifiers may not be keywords such as **`import`**, **`int`**, **`double`**, **`public`** or **`class`**. They must start with a letter, underscore character (_) or a dollar sign($). After the first character, you can include any combination of letters or numbers. Java is case sensitive, so `Sum`, `sum` and `SUM` are distinct (different) identifiers.

It is good programming practice to use descriptive and meaningful identifiers in your programs. By doing so it will make your programs easier to follow and understand. You should resist the temptation to use a short identifier simply to save a few keystrokes. For example if you wanted to keep track of the rate of pay for each of your employees, you could use an identifier such as `rateOfPay`. This is more meaningful than using shortened identifiers such as `r`, `rate`, or `rop`. Do not use “i”, unless it is an index.

Since we will be using meaningful identifier names, you may find your variable names include several words joined together. To make it easier to separate the words, use the following guidelines:

When naming variables or methods:

- The first letter of the variable name is lowercase
- Each subsequent word in the variable name begins with a capital letter.
- All other letters are lowercase.

This form is sometimes known as camel notation. For example:

```
int noOfStudents;              String lastName;
double totalMonthlySales;      char tryAgain;
```

Even though we use the same convention for both variables and methods, we can easily distinguish between the two since method names are always followed by parentheses even if they have no parameters. For example: `nextInt()` or `nextDouble()`.

When naming classes:

Follow the same convention used for variables and methods, except that the first letter of the variable name is uppercase. For example:

```
public class BouncingBall
public class RationalNumber
```

When naming constants (final):

Since constants do not change their value we want to distinguish them from variables. Therefore names of constants are in uppercase. If a name consists of several words, these are usually separated by an underscore(_). For example:

```
     final int MAX_SIZE = 1000;
     final double HST_RATE = 0.13;
```

A.2 Program Comments

Internal comments should be used in your Java programs to explain the code and make it easier for other programmers to understand. Also, when studying for tests and exams, comments will help you understand your own programs.

In Java you have three types of comments:

```
// The first style of comment begins with two /'s and then
// continues to the end of the line.  This type of comment
// is preferred for single line comments because you don't
// have to worry about closing off the comment

/*
The second style of comment uses the / and the * to start the
comment and the * and / to finish the comment. It is good for
multiple line comments and for commenting out a section of code
when you are testing your program.  If you use this type of
comment don't forget the * and / at the end.
*/

/** The third style of comment is very similar to the second
  * style.  the extra * on the first line is added to show that
  * this is a javadoc comment.  We will be using these comments
  * for our program's introductory comments and when commenting
  * method descriptions.  By using this style and the @ options
  * you can use javadoc to automatically generate HTML
  * documentation. Some of the special @ options we will be using
  * are shown below (see online help for a complete list):
  * @author
  * @version
  * @param
  * @return
  * @throws
  */
```

The following comments are a minimum requirement in your Java programs in this course:

1) Each class should include introductory java doc comments that include a clear description of the class and an `@author` tag with your name (not your student number) and an `@version` tag with the date the code was last modified. For example:

```
/** Keeps track of the rank and suit of a standard playing
  * Card.  Includes methods to construct a new Card, to
  * check if two Cards are equal, to compare two Cards by
  * both rank and suit and to display each Card
  * @author G. Ridout
  * @version Date: September 15, 2014
 */
```

2) Each method should include introductory java doc comments that include a description of what the method does, an `@param` for each parameter and an `@return` tag or `@throws` tag if the method returns a value or throws an Exception. Do not include the variable type in the `@param` (only the name) and do not include a variable name in the `@return`. There should only be one `@return` per method. You can include multiple `@throws` if you throw more than one type of Exception. In some cases you may want to list any preconditions (what should be true before the method is called) and postconditions (what will be true after the method is called) for your method. For example, the following comments would be for a `pow()` method:

```
/** Returns the value of the base raised to the given exponent
  * @param base     the base of the power
  * @param exponent the exponent to raise the base to
  * @return         the power (base raised to exponent)
*/
```

Some parts of this comment may seem redundant but it is important that all parts are included. If in doubt about the number of comments you should put into your program, check with your teacher.

3) For more complicated methods you should include comments to <u>clearly describe each section</u> of code in the body of the method to help make the code easier to follow. In most cases, comments should be above (<u>not</u> beside) the code. You do not need to comment each line of the program. With high-level languages such as Java, your code should be written so that it is easy to follow. Comments on sections of code help you and other programmers scan the code quickly. For example:

```
// Find the highest number in a list of integers
int highestNumber = listOfNumbers[0];
for (int index = 1; index < listOfNumbers.length; index++)
{
   if (listOfNumbers[index] > highestNumber)
      highestNumber = listOfNumbers[index];
}
```

In this case you do not need to comment the code within the loop since it is easy to understand once you know the purpose of the section of code. Never comment the obvious. For example

```
    int number = keyBrd.nextInt();     // Reads in an integer number
```

In this case the comment just restates what the line does which is pretty obvious from the original code. These types of comments are not necessary.

In Eclipse, you can use the Javadoc window to check your comments. The Javadoc window can also be used to show help for methods in other classes including Java’s standard classes.

Read over (proof read) your comments to make sure that they make sense and that they are grammatically correct with no spelling mistakes. **Note:** Eclipse will highlight spelling mistakes in your comments. Finally, sentences in your comments should start with an uppercase letter.

A.3 Code Paragraphing and Format

Spaces, indenting and blank lines can be used to make your Java code more readable from a programmer's point of view.

Indenting/Formatting Your Code

To help identify different structures (loops and selections) within your program code you should use indents and spaces to highlight these structures. By doing so, you will make your code easier to follow and debug. When using the Eclipse IDE we can press `Ctrl+Shift+F` to automatically format your code. If pressing `Ctrl+Shift+F` doesn't indent your code properly, your code may contain errors or extra blank lines.

Blank Lines

You should add blank lines to separate sections of code to make it easier to follow and debug your code. For example:

```
Scanner keyboard = new Scanner(System.in);

// Read in the hours worked and the rate of pay
System.out.print("Please enter your hours worked: ");
double hoursWorked = keyboard.nextDouble();
System.out.print("Please enter your rate of pay: ");
double rateOfPay = keyboard.nextDouble();

// Calculate and display the Gross Pay
double grossPay = hoursWorked * rateOfPay;
System.out.print("Gross pay: %.2f%n", grossPay);
```

Appendix B – Programming Guidelines and Tips

B.1 Properties of an Excellent Computer Program

Solves a Problem

- meets the user's needs (includes all of the features requested by the user)
- does what it is supposed to (free of any bugs or malicious code)
- works for all possible cases (checked with a complete set of test data)

User-friendly

- clear and easy to follow screen displays and printed output
- appropriate use of graphics, colours and sound
- bullet-proof (can handle user input mistakes)
- appropriate prompts and feedback
- consistent look and feel
- includes built-in help

Nicely Written Source Code

- uses descriptive names for variables, methods, ...
- simple, consistent and easy to follow
- commented and paragraphed (indented) as required
- easy to update/modify
- uses a modular design (methods and objects) to help cut down on duplicate code and to make the program more organized and readable. Also, makes the code easier to debug.

Efficient

- fast (runs quickly without delays)
- requires minimum system resources (memory, processor, ...)
- small code size with no unnecessary code

Flexible/Versatile

- can handle different situations (solves a general problem rather than a specific problem)
- compatible with other programs (running together and sharing data)
- allows the user to configure the program for their personal preferences (includes default settings)
- easy to upgrade (compatibility between versions)
- works on a variety of platforms

B.2 Programming Tips

Here are some programming tips that you should consider for every assignment:

1) Read the problem very carefully to make sure that you are solving the correct problem.
2) Think before you code! If you write perfect code for an incorrect or inefficient algorithm, your program will still be incorrect or inefficient.
3) Look over the warnings highlighted in your code and listed in the Problems window in Eclipse. In many cases the warnings are hints of potential errors in your code.
4) Look over your program code to make sure it is simple and easy to follow. Good code, even for complicated algorithms, should be easy to follow without any unnecessary code.
5) Spend some time testing your code to make sure it can handle a variety of test cases. Make sure that you test the boundary cases (minimum and maximum values for each input). Check special cases (0, 1, negative values). Read the problem carefully. If the problem states that you can assume an inputted value will be positive, then you don't need to handle non-positive input. See section 1.3 for more on testing your code.
6) Be careful of integer overflow when testing large values. Java will not give you an error when an integer overflows – it will just do the wrong thing. Think carefully before changing all your variables to `long` to avoid integer overflow.
7) Look at your inequalities carefully – do you want "<" or "<=". In many cases, the results will be quite different.
8) In methods, take advantage of the fact that a return statement will quit the method immediately, making your code run faster. Do not use `break` or `continue` statements.
9) Know the Java language – use while loops and for loops appropriately (avoid do while loops in most cases). In if statements, use the else option when appropriate.
10) In order to maximize your learning experience, assignments should be completed on your own. Try to avoid the temptation to look at your neighbour's solution.

B.3 Eclipse Tips

Every class or program in Eclipse needs to be part of a project. When you add code to a project you need to make sure that the project knows about this new code. Therefore, it is important that you **do not** just add code to a project using File Explorer in Windows. Instead you should drag and drop any source code into the `src` folder in the Eclipse IDE.

To transfer an existing project from one workspace (e.g. home workspace) to another workspace (e.g. school workspace), it is important to use the `Import` feature in Eclipse. You should use the `File->Import->General->Existing Projects into Workspace` option from the top menu.

To enhance your coding experience with Eclipse, you should become familiar with the following Eclipse features:

Code Completion
Automatically suggests possible code as you type. Use Ctrl-Space if it doesn't come up. Offers suggestions based on your current code.

Quick Fix
When you get an error in your code (Red X in the left margin), you can use "quick fix" to get suggestions on how to fix this error. To bring up the quick fix suggestions, hover over your error. In most cases the first suggested quick fix will fix the problem but sometimes you will have to look at a choice down the list. In some cases none of the suggested quick fixes will fix the problem. So, think before you select.

Refactoring
Allows you to update your code without changing its function. This is a very effective way of renaming variables and methods.

Templates
Offers possible templates for certain commands such as: `for` loops or `if` statements. Use Ctrl+space to bring up the suggested templates.

Automatic Typing
Automatically adds in required code. For example: closing `}` or `"`. Will also add in `*`'s and `@ tags` for javadoc comments (`/**`).

Comments Spell Check
Checks for spelling mistakes in your comments. Use this feature!

Hover Help
Gives you help for a method when you hover the cursor over the method name.

Javadoc View Window
Shows help for any method with javadoc comments including any methods that you wrote yourself. Only works if your javadoc comments are formatted correctly.

Debugger
Helps to debug your code by allowing you to set breakpoints, step through your code and to look at the value of your variables as your code runs. We will be covering how to use the debugger in class.

Appendix C – Big O Notation and Algorithm Run Times

In order to compare the runtime efficiency of different algorithms we can use “Big O” notation (pronounced Big Oh). This notation gives an estimate of an algorithm’s run time as a function of the size of the problem being analyzed. The following is an informal non-mathematical discussion of Big O notation.

Assume that we want to process n pieces of data and we find that the number of required operations and therefore the corresponding runtime is a function of n. For example: $2n^2 + 3n + 4$. Since we just want an estimate of the algorithm’s run time we want to simplify this expression. First we note that for large values of n, the lower order terms $3n$ and 4 would be insignificant compared to the $2n^2$ term. Next we note that the value of any coefficient of the n^2 term will lose its significance depending on the hardware’s processing speed for each operation. So in simplest terms, we can say that the run time grows at the order of n^2. Using the Big O notation, we would say the run-time is $O(n^2)$. With an $O(n^2)$algorithm, if we increase the size of our problem by a factor of n, the runtime would increase by a factor of n^2.

When you are trying to determine the Big O runtime of an algorithm based on the number of required operations as a function of n, you should:

1) Only consider the dominant term as n gets very large. For example n^3 is more significant than n^2 and 2^n is more significant than n^3 as n gets large.
2) Don’t worry about any constant coefficients when giving the Big O run time. For example, algorithms with runtimes of $0.4n^2$ and $1.5n^2$ are both $O(n^2)$.

As was mentioned earlier, this is a very simple non-mathematical discussion of Big O notation. A more formal mathematical definition of Big O can be found on-line.

Let’s look at a section of code and try to determine its runtime using Big O notation. We will assume the size of the array is n.

```
// Double the value of each element in an array of ints
for (int i = 0; i < numbers.length; i++)
   numbers[i] *= 2;
```

Since the loop runs n times, the runtime of this section of code would be proportional to n which means the runtime would be: kn, where k is a constant determined by how fast the hardware can compare and increment the loop counter and multiply each array element by 2. Since we ignore any constant coefficients when finding the Big O runtime, we don’t need to know the actual value of k. Therefore the runtime of the given section of code would be $O(n)$ in Big O notation.

If we only processed every other element of the array (changing `i++` to `i+=2`), the loop would run only n/2 times and now the runtime would be: ½kn. But, once again, since we ignore the constant coefficient when using Big O notation, the runtime would still be $O(n)$.

However, if we changed `i++` to `i*=2`, the runtime would be `O(log n)` since the number of times the loop repeats would be approximately $\log_2 n$ and the runtime would be: $k \log_2 n$. Even though the number of times the loop repeats is `log` to the base 2, in Big O notation we would say `log n`, since:

$$k\log_2 n = \frac{k\log_{10} n}{\log_{10} 2} = k_1 \log_{10} n \qquad \text{where } k_1 = \frac{k}{\log_{10} 2}$$

and since we ignore the constant coefficients in Big O notation it doesn't matter whether we ignore the original k value or k_1, which is also just a constant. However, if we are asked to find the actual number of times that the loop repeats we should use the $\log_2 n$ value.

Let's try another example. Below is the code for a simple bubble sort. Again, we assume the number of elements in the array we are sorting is n.

```
// Sort the elements of an int array using a simple bubble sort
for (int upperLimit = list.length - 1; upperLimit > 0; upperLimit--)
{
   for (int check = 0; check < upperLimit; check++)
   {
      // Compare adjacent pairs
      if (list[check]  > list[check + 1])
      {
         // Swap the out of order elements
         int temp = list[check];
         list[check] = list[check+1];
         list[check+1] = temp;
      }
}
```

Looking at this code we can see that the outer loop is repeated `n-1` times. The inner loop would then run `n-1` times the first time, `n-2` the second time, down to `1` when checking the last pair. Therefore on average the inner loop runs `n/2` times each time so the code inside the inner loop would run a total of:

$$(n-1) \times \frac{n}{2} = \frac{n^2}{2} - \frac{n}{2} \text{ times}$$

Since the code inside the inner loop is where we do most of the work including comparing pairs of elements and swapping elements that are not in order, the runtime of the sort would be proportional to the number of times the inner loop runs. Therefore, ignoring the n term (which would be less significant as n get larger) and any constant coefficients of the n^2 term, we could say this bubble sort algorithm is $O(n^2)$.

We will be looking at Big O runtimes for various algorithms throughout the course. On the next page are some practice questions relating to Big O notation.

1) Given the following run times as a function of n (the number of elements being processed), what would be the run time using Big O notation.

a) $0.5n^3 + 12n^2 + 2n + 7$
b) $(n+2)^4$
c) $(n+1)\left(\frac{1}{n}+2\right)(n^2+3)$
d) $3^n + 2n^3 + 3$
e) $\sqrt{n} + 2n + 9$
f) $4\log_2 n + 3\log_3 n + 2\log_4 n$
g) $n\log_2 n + \mathrm{n}^2\log_2 n + 5\log_2 n$

2) Rank the following Big O run times from the fastest to the slowest.

$O(n \log n)$	$O(n)$
$O(n^2)$	$O(n^3)$
$O(\sqrt{n})$	$O(1)$
$O(n^2 \log n)$	$O(2^n)$
$O(\sqrt{n}\log n)$	$O(2^{\log n})$
$O(3^n)$	$O((\log n)^2)$

3) Find the run time of each of the following <u>sections</u> of code using Big O notation.

```
int result = 0;
for (int i = 0; i < n/2; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k+=2)
         result += i+ 2*j + 3*k;
```

```
int result = 0;
for (int i = 1; i < n; i*=2)
   for (int j = 0; j < n; j++)
      result += i+ 2*j;
```

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      numbers.add(0, i*j);
```

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i < n; i++)
    numbers.add(i);
for (int j = 0; j < n; j++)
    numbers.add(numbers.remove(numbers.size() - 1)* j);
```

Appendix D – StringBuilder and ArrayList Methods

D.1 **StringBuilder methods**

Constructors

`StringBuilder()`

Constructs an empty `StringBuilder` with an initial capacity of 16 characters.

`StringBuilder (int length)`

Constructs an empty `StringBuilder` with an initial capacity specified by the `length`.

`StringBuilder (String str)`

Constructs a `StringBuilder` with the initial contents the same as the argument `str`. The initial capacity of the new `StringBuilder` will be `str.length() + 16`.

Methods

`StringBuilder append(Object objectToAdd)`

Appends the string representation of the `objectToAdd` to the `StringBuilder`. `objectToAdd` could be an object (e.g. `String`) or a primitive (e.g. `double`).

`int` `capacity()`

Returns the current capacity of the `StringBuilder`.

`StringBuilder delete(`**`int`**` start, `**`int`**` end)`

Removes the characters from `start` (inclusive) to `end` (exclusive).

`StringBuilder deleteCharAt(`**`int`**` index)`

Removes the character at the specified position in this `StringBuilder`.

`void` `ensureCapacity(`**`int`**` minimumCapacity)`

Ensures that the capacity of the builder is at least equal to the specified minimum.

`StringBuilder insert(`**`int`**` insertIndex, Object objectToInsert)`

Inserts the string representation of the `objectToInsert` into the `StringBuilder` at the `insertIndex`. The `objectToInsert` could be an object or a primitive.

`StringBuilder replace(`**`int`**` start, `**`int`**` end, String str)`

Replaces the characters in this `StringBuilder` between `start` (inclusive) and `end` (exclusive) with the characters in the string argument `str`.

`StringBuilder reverse()`

Reverses the order of the characters in this `StringBuilder`.

`void` `setCharAt(`**`int`**` index, `**`char`**` ch)`

Sets the character at the specified index of this `StringBuilder` to `ch`.

`String toString()`

Returns a string representing the data in this `StringBuilder`.

String methods also available for `StringBuilders` include: `charAt()`, `indexOf()`, `lastIndexOf()`, `length()`, and `substring()`. Note the `equals()` method <u>does not</u> compare the contents of two `StringBuilders`. To compare two `StringBuilders` use `toString` and then compare the resulting `Strings`.

D.2 `ArrayList` methods

For the following code, `E` indicates the type of elements held in the `ArrayList`.

Constructors

`ArrayList()`
Constructs an empty list with an initial capacity of 10.

`ArrayList(Collection<E> c)`
Constructs a list containing the elements of the specified collection.

`ArrayList(int initialCapacity)`
Constructs an empty list with the specified initial capacity.

Methods

`int` `size()`
Returns the number of elements in this `List`.

`boolean` `add(E element)`
Appends the specified element to the end of this `List`. Returns `true` if added.

`void` `add (`**`int`** `index, E element)`
Inserts the given element at the specified position in this `List`. (`0<= index <= size`). Moves elements at position index and higher to the right and updates the size.

`E get(`**`int`** `index)`
Returns the element at the specified position in this `List`.

`E set(`**`int`** `index, E element)`
Replaces the element at the position `index` with the specified element. Returns the element formerly at the specified position.

`E remove(`**`int`** `index)`
Removes the element at the position `index`, moving elements at position `index + 1` and higher to the left and updates the size. Returns the element formerly at the specified position.

`int` `indexOf(E element)`
Searches for the first occurrence of the given argument in this `List`, testing for equality using the `equals` method. Returns -1 if the item is not found.

`int` `lastIndexOf(E element)`
Searches for the last occurrence of the given argument in this `List`, testing for equality using the `equals` method. Returns -1 if the item is not found.

`boolean` `contains(E element)`
Returns `true` if this `List` contains the specified element, otherwise returns `false`.

`void` `clear()`
Removes all of the elements from this `List`.

`List<E> subList(`**`int`** `fromIndex,` **`int`** `toIndex)`
Returns a view of the portion of this `List` between the specified `fromIndex`, inclusive, and `toIndex`, exclusive

`boolean` `isEmpty()`
Tests if this `List` has no elements.

Appendix E – File Input and Output

The following sections cover reading from text files using either a `Scanner` (more convenient) or a `BufferedReader` (faster), writing to text files using a `PrintWriter`, reading objects from a file all in one step using an `ObjectInputStream` and writing objects to a file using an `ObjectOutputStream`.

E.1 Using a `Scanner` to read from a text file

To read data from a text file we can use a `Scanner` object created with a `File` object parameter instead of the `System.in` parameter used for keyboard input. For example to read and display all of the lines in a text file, we can use the following code:

```
Scanner inFile = new Scanner(new File("data.txt"));
while (inFile.hasNextLine())
{
   String nextLine = inFile.nextLine();
   System.out.println(nextLine);
}
inFile.close();  // Close the file asap
```

The `hasNextLine` method returns `true` if there is still data left in the file. The `nextLine` method reads the text file line by line. We can also use the `Scanner` methods `next`, `nextInt`, `nextDouble`, `nextLong` etc. to read individual elements from the file one at a time. If we don't know how much data is in the file, we can use the methods: `hasNext`, `hasNextInt`, `hasNextDouble`, `hasNextLong` etc. to check if there is still data left in the file.

When opening the file, if the given file doesn't exist, a `FileNotFoundException` will be thrown. To deal with this checked exception, you need to include **`throws`** `FileNotFoundException` at the end of the `main` method heading. For example:

```
public static void main (String[] args) throws FileNotFoundException
```

In this case, if there is an error opening the file we would just ignore the error and our program would terminate with a run-time error. This is a simple way of dealing with a checked exception. In Appendix G we will see how we can use a `try catch` block to properly deal with checked exceptions. Note: To use the `FileNotFoundException` and `File` classes you will also need to include: **`import`** `java.io.*;` at the top of your program.

Here is another example that reads a simple text file used to track of the top player and his/her score for a game. This code assumes the text file would contain the name of the player on the first line and their score on the second line. For example, assume the following data is in a text file called `"topScores.txt"`:

```
Jim Nasium
3456
```

```
// Open the file for input
// "topScores.txt" is the name of the file
Scanner fileIn = new Scanner (new File ("topScores.txt"));

// Read in the topPlayer and his/her score
// This is very similar to keyboard input
String topPlayer = fileIn.nextLine();
int topScore = fileIn.nextInt();

// Close the file
fileIn.close();
```

E.2 Using a `BufferedReader` to read from a text file

Using a `Scanner` to read from a text file is very convenient since you can use methods such as `nextInt` and `nextDouble` to read numbers from the file even if the numbers are not on the same line. However, since `Scanners` can be quite slow, for larger text files we should use a `BufferedReader` which is much faster. Below is the code to find the total of a file of integer numbers using a `BufferedReader`.

We will assume that we don't know how many numbers are in the file so we will keep reading the file until the `readline` method returns a `null` value which indicates the end of file. One disadvantage of `BufferedReaders` is that they don't have specific methods to read integers and doubles etc. so we need to read each line as a `String` and then convert it to a number using `Integer.parseInt` or `Double.parseDouble`.

```
// Since this is large file, the total could get big
long total = 0;

// Open the file
BufferedReader fileIn =
           new BufferedReader(new FileReader("numbers.txt"));

// Keep reading the file until null is returned (end of file)
String line;
while ((line = fileIn.readLine()) != null)
{
    // Need to convert and add in each number
    int next = Integer.parseInt(line);
    total += next;
}

// Close the file and display the total
fileIn.close();
System.out.println(total);
```

This code assumes that the file contains one number per line. If the file contained more than one number on each line, we would need to use a `StringTokenizer` (see section 5.9 in the Grade 11 course pack) to break down each line.

Note: The code on the previous page can throw an `IOException` so we need to use a `try…catch` block or add **`throws`** `IOException` to the heading of our `main` program (see example in `E.1`). The `IOException` class is a superclass of the `FileNotFoundException` so if you include **`throws`** `IOException` you don't need to include a **`throws`** `FileNotFoundException` as well.

To use the `BufferedReader, FileReader` and the `IOException` classes you will also need to import each class or include **`import`** `java.io.*;` at the top of your program to import them all at once.

E.3 Using a `PrintWriter` to write to a text file

To write data to a text file we can use a `PrintWriter` object and the `print`, `println` and `printf` methods. Writing to a text file is very similar to writing to the System console using `System.out.println`.

Here is an example section of code that opens up a text file and then writes the name of the top player and his/her score to this file. This code will create the same file discussed in sections E.1. Note: In this code we used `println` but we could have also used `printf` if we wanted formatted output.

```
// Opens the file for output
// "topScores.txt" is the name of the file
PrintWriter fileOut =
               new PrintWriter(new FileWriter("topScores.txt"));

// Writes the top players name and score to the file
// Assumes topPlayer and topScore have been defined
// Writes one value per line (makes it easier to read in)
fileOut.println(topPlayer);
fileOut.println(topScore);

// Close the file as soon as you are finished with it
fileOut.close();
```

Note: The above code can throw an `IOException` so we will need to use a `try…catch` block or add **`throws`** `IOException` to the heading of our `main` program (see example in `E.1`). The `IOException` class is a superclass of the `FileNotFoundException` so if you include **`throws`** `IOException` you don't need to include a **`throws`** `FileNotFoundException` as well.

To use the `PrintWriter, FileWriter` and the `IOException` classes you will also need to import each class or include **`import`** `java.io.*;` at the top of your program to import them all at once.

E.4 Reading and Writing Objects to a File

If you want to read or write an object to a file you could write each instance variable to a text file individually or you could write the whole object to a file all in one step. We will demonstrate the second option with a `Statistics` class that keeps track of the statistics (e.g. number of games, the number of wins, longest streak, high score etc.) for a game program. Here is partial code for this class.

```
public class Statistics implements Serializable
{
   // instance variables to keep track of games statistics
   // methods to process these statistics including a
   // constructor to create a new Statistics object with
   // all statistics reset to zero
}
```

Since we are going to be writing our `Statistic` object to a file, we need to implement the `Serializable` interface. Since our instance variables will be primitives or Strings no extra code is needed to become a `Serializable` object. However, you could add a default `serialVersionUID` if you wish.

To write a whole `Statistics` object to a file all in one step we add the following method to our `Statistics` class. You could also put this code in an `update` method since you would probably want to write to the file every time you updated the statistics.

```
public void writeToFile(String fileName)
{
   // Since we may have trouble writing to the file, we
   // should include a try catch block to catch any errors
   try
   {
      // Write the entire Statistics object to a file
      ObjectOutputStream fileOut =
          new ObjectOutputStream(new FileOutputStream(fileName));
      fileOut.writeObject(this);
      fileOut.close();
   }
   catch (IOException exp)
   {
      System.out.println("Error writing to the file");
   }
}
```

When you write an object to a file it keeps track of the object's information including the current value of any variables. The data is not stored like a text file so you usually can't read the data in the file outside of your program. With this in mind don't use a ".txt" extension in the file name. Instead, you could use a ".dat" extension.

To read the file we need to create a method that will read and return the `Statistic` object data from the file. To do this we can create a static method in the `Statistics` class that returns a `Statistics` object. Usually we try to minimize the number of static methods in a class but in this case the static method will keep all of the `Statistics` class code together. Here is the code:

```
public static Statistics readFromFile(String fileName)
{
   // Once again we need a try catch block.  In this case
   // we could get a variety of exceptions, but since
   // all exceptions mean that we can't read the file
   // properly we will lump them into one catch block
   try
   {
      // Try to open the file and read in the statistics
      // information.  Read the entire Statistics object from
      // a file.  Since readObject returns a reference to an
      // Object we need to cast it to a Statistics object
      ObjectInputStream fileIn =
          new ObjectInputStream(new FileInputStream(fileName));
      Statistics stats = (Statistics )fileIn.readObject();
      fileIn.close();
      return stats;
   }
   // This could include different types of Exceptions
   catch (Exception exp)
   {
      // If we had trouble reading the file (e.g. it doesn't
      // exist or if our file has errors) an Exception will be
      // thrown and we can create and return a new Statistics
      // object with all values reset. This assumes we have a
      // a constructor to do this.
      return new Statistics();
   }
}
```

In the above method we have included only one catch block which catches different kinds of exceptions. Usually we want multiple catch blocks to catch each type of exception but in this case, since all exceptions will be caused by a problem reading in the file (e.g. wrong file name, bad data structure in the file etc.), we have put them all together.

To call this method in our main program we would use:

```
Statistics myStats = Statistics.readFromFile("stats.dat");
```

Reading and writing objects all in one piece is a convenient way to store object data in a file. Reading and writing other objects including an entire `ArrayList` of objects is just as easy. Unfortunately not all object data (e.g. Images) can be written to a file.

Appendix F – `BigInteger` Methods

A partial list of the most useful `BigInteger` methods is given below with an example of how to use each method. In each example, all of the numbers are `BigIntegers`. For more details and additional `BigInteger` methods, see the Java API.

`abs()`

    e.g. absValue = number.abs()   // absValue = |number|

`add(BigInteger anotherBigInt)`

    e.g. sum = first.add(second)   // sum = first + second

`compareTo(BigInteger anotherBigInt)`

    e.g. if (first.compareTo(second) > 0)
                c.println("The first number is larger")

`divide(BigInteger anotherBigInt)`

    e.g. div = first.divide(second)   // div = first/second

`equals(Object anotherObject)`

    e.g. if (first.equals(second))
                c.println("The numbers are equal")

`max(BigInteger anotherBigInt)`

    e.g. largest = first.max(second)

`min(BigInteger anotherBigInt)`

    e.g. smallest = first.min(second)

`multiply(BigInteger anotherBigInt)`

    e.g. product = first.multiply(second) // product = first × second

`pow(int exponent)`

e.g. `power = base.pow(exponent)`   `// power =` $\texttt{base}^{\texttt{exponent}}$

`remainder(BigInteger anotherBigInt)`

    e.g. rem = first.remainder(second)   // rem = first % second

`subtract(BigInteger anotherBigInt)`

    e.g. diff = first.subtract(second)   // diff = first - second

`valueOf(long value)`

A static method that converts a `long` or an `int` number to a `BigInteger`

    e.g. number = BigInteger.valueOf(25)

`toString()`

Converts a `BigInteger` number to a String

Appendix G -- Exceptions and Exception Handling

The Java language reports run-time errors by throwing exceptions. An exception is a special type of object that reports on certain run-time errors. To handle an exception you write code that "catches" these "thrown" exceptions. For example, the following section of code is part of the code for a `LetterGrid` class that keeps track of a 2D array of characters for a Word Search or Boggle program.

```
public class LetterGrid
{
  private char[][] grid;

  public LetterGrid(String fileName)
  {
    try
    {
      Scanner inFile = new Scanner(new File(fileName));
      ArrayList<String> lines = new ArrayList<String>();
      while (inFile.hasNextLine())
        lines.add(inFile.nextLine());
      grid = new char[lines.size()][];
      int row = 0;
      for (String nextLine : lines)
      {
        grid[row] = nextLine.toCharArray();
        row++;
      }
      inFile.close();
    }
    catch (FileNotFoundException exp)
    {
      grid = new char[][] {{'B', 'A', 'D'},
                           {'F', 'I', 'L', 'E'},
                           {'N', 'A', 'M', 'E'}};
    }
  }
```

In this example, we are reading the data for our `LetterGrid` object from a text file. Since opening a file can throw a `FileNotFoundException` we need to deal with this exception. We could just re-throw the exception but it is better practice to deal with the exception ourselves. The section of code in the curly braces after the `try` statement is called the try block. The computer tries to run each statement in the try block. If one of these statements throws an exception (causes an error) the program immediately jumps out of the try block and looks for a `catch` statement that catches the exception just thrown.

For example, if the file name given to the constructor is invalid, when we try to open the `Scanner` a `FileNotFoundException` would be thrown. The `catch` block can catch a `FileNotFoundException` (indicated in the parentheses after the keyword `catch`) so, when this exception is thrown, program control is passed to the first line in this `catch` block. This would create a default LetterGrid that will display as:

```
BAD
FILE
NAME
```

to show the user there is an error.

You can have many catch blocks following a try block to catch and deal with different types of exceptions. If your catch blocks do not catch a particular exception it will be passed along (re-thrown) for some other section of code to deal with (catch). In most cases you should deal with exceptions as soon as possible.

On the other hand, if the file name was valid, no exception would be thrown and all of the statements in the try block would have been completed, reading in the file data. When the try block is successfully completed, control passes to the first statement after the last catch block.

The `FileNotFoundException` in the above code is an example of a “checked” exception. In Java, you always have to write code to deal with any checked exceptions. Some other exceptions such as `ArrayIndexOutOfBoundsException` or `NumberFormatException` are examples of “unchecked” exceptions. In Java, you do not have to write code to deal with unchecked exceptions. However, if this exception gets thrown a message will be reported in an error window and the program may stop running. In most cases, if an unchecked exception is thrown, there is usually something wrong with your code. For small practice problems you don't have to deal with unchecked exceptions, however when you are writing code that will be used by others, you should anticipate and deal with all exceptions.

We can also throw exceptions in our own code if we want to report on errors. For example, when we are writing our own methods, if the value of a parameter sent to a method is invalid we can throw an `IllegalArgumentException`.

References

Bloch, Joshua
Effective Java, Programming Language Guide
Sun Microsystems, Inc, Palo Alto, CA 2001

Burnette, Ed
Eclipse IDE Pocket Guide
O'Reilly, Sebastopol, CA 2005

Dasgupta, S, Papadimitriou, C. H. and Vazirani, U. V.
Algorithms
McGraw-Hill Science/Engineering/Math 2006

Halim, Steven and Halim, Felix
Competitive Programming 3
Lulu, Raleigh, NC 2013

Horstmann, Cay S. and Cornell, Gary
Big Java, 3rd Edition
John Wiley & Sons, Hoboken, NJ 2008

McConnell, Steven C.
Code Complete
Microsoft Press, Redmond, WA 1993
A second edition (2004) is also available

Oracle's Java web site including the Java help files and tutorials
The latest version of the Java API can be found at:
    `http://docs.oracle.com/javase/8/docs/api/`
The latest version of the Java tutorial can be found at:
    `http://docs.oracle.com/javase/tutorial/`
The latest version of the Java Virtual Machine Specification can be found at:
    `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`

Ridout, Gord
Introduction to Programming with Java
Version 3.1 2014

Skiena, Steven S.
The Algorithm Design Manual (2nd Edition)
Springer-Verlag London Limited, London, England 2008

USA Computing Olympiad
`http://www.usaco.org/`

UVa Online Judge - Universidad de Valladolid
`http://uva.onlinejudge.org/`