CS 2440
Computer Science II

Dr. Alice McRae
Dr. Dee Parks
Professors, Computer Science
Appalachian State University

Table of Contents

Introduction

These worksheets are meant to supplement, not replace, the course textbook. You should use the worksheets by bringing them to class with you and taking notes on them. Each day, your instructor will put a worksheet on the screen and use it as a guide to the discussion. She will write on her copy as the discussion proceeds, and you should write on yours. Later, when you are studying, you can go back over the worksheets and make sure you fully understand the topics that were discussed.

You are encouraged to study with one or more other students in the course. Since each of you will have the same worksheets, you will be able to more easily navigate the different topics that are covered. You can refer to the page numbers and direct everyone in the group to the same material.

Your instructor would appreciate any suggestions that you have for improving these worksheets.

Enumerated Types

Enumerated types are simple types that you create to hold a limited number of values. The simplest way to declare an enumerated type in Java is to use the keyword `enum` followed by an identifier (by convention, start it with an uppercase letter) and a list of values, set off in curly braces and delimited by commas. Place a semicolon at the end. The values are not `Strings` (no quotes), and are treated like constants. By convention, use all uppercase for the values.

Here is an example of an enumerated type named `Month` with values describing each of the twelve months. The enumerated type is placed within a class that makes use of it.

```
public class SomeClassUsingMonths
{
    public enum Month {
        JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
        JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
    }

    // more code
}
```

Now you write an enumerated type for days of the week:

public enum day {
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

Once an enumerated type has been declared, you can declare fields, parameters, or local variables of that type. Here are variables of type `Month`:

```
    private Month favoriteMonth;
    private Month vacationMonth;
    private Month rainyMonth;
```

If the `enum` is public, and you want to declare variables of that type outside of the class, you need to tell Java what class to find the `enum` in.

```
public class SomeOtherClass
{
    private SomeClassUsingMonths.Month birthdayMonth;

    // some other code
}
```

To assign one of the enumerated values to a variable of the enumerated type, use the `enum` name and a dot before the value.

```
    // constructor
    public  SomeClassUsingMonths(Month favorite)
    {
        favoriteMonth = favorite;
        vacationMonth = Month.JUNE;
        rainyMonth = Month.APRIL;
    }
```

As you might suspect, you can declare and initialize a local variable in one step.

```
        Month temp = Month.MARCH;
```

Now you declare a variable of your days of the week type and then assign it the value `Wednesday`.

favoriteDay = Day.WEDNESDAY;

For every enumerated type, there is a method called `values` that returns an array whose elements are the values of the enumerated types in the order in which they are listed in the definition.

```
   public static void main(String[] args)
   {
       Month[] monthArray = Month.values();

       for (Month m : monthArray)
       {
           System.out.println(m);
       }
   }
```

The above code would produce this output:

```
JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER
```

All enumerated types automatically extend the `Enum` class. Thus all enumerated types inherit several useful methods. We just saw one of them, `values.` Others are `compareTo`, `equals`, `ordinal`, and `toString`.

This table shows descriptions of these methods taken from the Java 8 API at http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html:

public final int compareTo(E o) Compares this enum with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are only comparable to other enum constants of the same enum type. The natural order implemented by this method is the order in which the constants are declared.
public final boolean equals(Object other) Returns true if the specified object is equal to this enum constant.
public final int ordinal() Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero). Most programmers will have no use for this method. It is designed for use by sophisticated enum-based data structures, such as `EnumSet` and `EnumMap`.
public String toString() Returns the name of this enum constant, as contained in the declaration. This method may be overridden, though it typically isn't necessary or desirable. An enum type should override this method when a more "programmer-friendly" string form exists.

You can compare two `enum` objects with == or with `equals` and get the same result.

```java
if (favoriteMonth == Month.JUNE)
    System.out.println ("That is my favorite, too.");
if (vacationMonth.equals(Month.AUGUST))
    System.out.println("I wish it was sooner!");
```

Java is a bit inconsistent in this next piece of information, but if you use enumerated types as the case values in a switch statement, you don't use the `enum` name and a dot with the case.

```java
switch (favoriteMonth) {
    case JANUARY:
        System.out.println ("Not in Boone.");
        break;
    case FEBRUARY:
        System.out.println ("Ah… Valentine Day's Month");
        break;
     // the rest
}
```

You can find more information on enumerated data types in the *Head First Java* book, available on Safari through the Library's web site, or online.

Two-Dimensional Arrays

There are two steps to the creation of a two-dimensional (2-d) array: (1) declaring a reference of the right type, and (2) allocating the actual array. The following code shows the declaration of three references to 2-d arrays. The first line declares a reference to a 2-d array of `ints`. The second line declares a reference to a 2-d array of `chars`, and the third declares a reference to a 2-d array of objects of some class called `Cell`.

```
int  [][] multiplicationTable;
char [][] wordSearchPuzzle;
Cell [][] mineSweeperGrid;
```

Show how to declare a reference to a 2-dimensional array of `Strings` called `wordtable`.

String [] [] wordtable;

Show how to declare a reference to a 2-dimensional array of `QuiltSquare` called `quilt`.

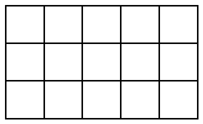
Quilt [][] QuiltSquare;

You can declare arrays with more than two dimensions if you choose:

```
int [][][] threeDTable;
```

When you declare a reference to a multi-dimensional array, you tell Java how many dimensions the array will have (by including a pair of square brackets for each dimension) but you do not specify the size of each of the dimensions. When you allocate the actual array, however, you must tell Java the size of each dimension.

We think of a 2-d array as a table made of cells that are organized into rows and columns. For example, here is how we picture a 2-d array with 3 rows and 5 columns:



Let's examine some code that allocates space for 2-d arrays. We specify the number of rows in the first set of square brackets, and the number of columns in the second set.

```
multiplicationTable = new int[10][10];
wordSearchPuzzle = new char[numRows][numColumns];

// create cells with borders around the edge
mineSweeperGrid = new Cell[rows+2][columns+2];
```

Create the `wordtable`, whose reference you declared before, and give it 3 rows and 4 columns.

Wordtable = new string[3][4];

Create the quilt of `QuiltSquare` so that is twice as long as it is wide. Use a variable called `width` that contains the desired number of squares in the width of the quilt. Make the length the first dimension.

QuiltSquare = new quilt[width][width * 2];

If we wish, we can create a multi-dimensional array one dimension at a time starting with the rows. We could, for example, create the `wordSearchPuzzle` array like this:

```
    // create an array to hold numRows char arrays
    wordSearchPuzzle = new char[numRows][];
    for (int i = 0; i < numRows; i++)
    {
        // create each row to hold numCols chars
        wordSearchPuzzle[i] = new char[numCols];
    }
```

Draw a picture to illustrate what a 2-d array is really like when it is allocated in memory. Your instructor will show you what to draw.

Using this technique, you can also create "jagged" arrays. The array allocated below has four rows. The first row has 6 cells, the next one has 10, the next one has 5, and the last one has 7. Draw a picture of the array below the code.

```
int[][] jaggedArray = new int[4][];

jaggedArray[0] = new int[6];
jaggedArray[1] = new int[10];
jaggedArray[2] = new int [5];
jaggedArray[3] = new int [7];
```

Look back at the `wordSearchPuzzle` definition above. The value of `wordSearchPuzzle.length` is the length of the first dimension (the number of rows). In this case, `wordSearchPuzzle.length` is whatever `numRows` was when the table was created. The value of `wordSearchPuzzle[3].length` would be the length of the third row (row numbers begin at 0), in this case `numCols`.

In the `jaggedArray` created above, what would be the value of `jaggedArray.length`? _______

What would be the value of `jaggedArray[2].length`? _________

Which of the following would be valid entries in the `jaggedArray` table? Circle valid ones.

```
jaggedArray[0][5]                jaggedArray[4][2]
jaggedArray[1][10]               jaggedArray[3][9]
jaggedArray[2][3]                jaggedArray[0][0]
```

You may initialize a 2-dimensional array when you declare it. Place each row's column values in braces, using a comma as a delimiter both between column values and rows.

```
String [][] words = {{"dog", "cat", "squirrel"}, {"pig", "goat", "elephant"}};
```

Draw the table, `words`, based on the initialization given:

Nested loops are often used to process a multi-dimensional array. In the following method, the outer for loop is advancing i over the rows, and the inner for loop is advancing j over the columns of each row. Show the results of the method on the `wordTable` drawn below:

```
public void fillTable(String[][] wordtable)
{
    for (int i = 0; i < wordtable.length; i++)
    {
        for (int j = 0; j < wordtable[ i ].length; j++)
        {
            if (i < j)
            {
                wordtable[i][j] = "cat";
            }
            else if (i > j)
            {
                wordtable[i][j] = "dog";
            }
            else
            {
                wordtable[i][j] = "ox";
            }
        }
    }
}
```

	j=0	j=1	j=2	j=3
i=0				
i=1				
i=2				

Show what the following method would print if we call it on the above array:

```
public void printTable(String[][] wordtable)
{
    for (int i = 0; i < wordtable.length; i++)
    {
        for (int j = 0; j < wordtable[i].length; j++)
        {
            System.out.println(wordtable[i][j]);
        }
    }
}
```

As you process two-dimensional tables with nested loops, consider the following placement of code.

```
public void processTable(String[][] table)
{
    // code to be done before or at the start of the whole table
    for (int i = 0; i < table.length; i++)
    {
        // code to be done at the start of each row
        for (int j = 0; j < table[i].length; j++)
        {
            // code to be done for each entry in the table
        }
        // code to be done at the end of each row
    }
    // code to be done after the table is processed
 }
```

Rewrite the `printTable` method above so that the output looks like a neat table with row and column headings.

If the code inside the loop gets complicated, create separate methods to handle parts of the job.

Here are some examples that illustrate how to send actual parameters to methods that handle parts of 2-dimensional arrays. The examples illustrate (1) how to send a reference to the entire table as an actual parameter to a method, (2) how to send a reference to a row as an actual parameter to a method, and (3) how to send the contents of a cell as an actual parameter to a method.

```
processTable(wordtable);          // send whole table

processRow(wordtable[2]);         // send row 2

processElement(wordTable[i][j]);  // process element wordTable[i][j]
```

Here are examples that show how to declare the formal parameters of methods that receive references (1) to entire 2-dimensional arrays, (2) to rows of 2-dimensional arrays (which are nothing more than single dimension arrays), and (3) to elements (which are nothing more than a variable of the array's type).

```
public void processTable(String[][] table)
{
    // method to process a whole table of Strings
}

public void processRow(String[] words)
{
    // method to process a single array (one row) of Strings
}

public void processElement(String word)
{
    // method to process a single String
}
```

Now let's write some methods that work with 2-dimensional arrays. Write an `int` method called `countPositives` that takes as an input parameter a 2-dimensional array of `int` and returns the count of all the positive integers (those > 0) in the array.

Write a method that prints a 2-dimensional array of integers as a table. At the end of each row, also print the average of the row. Put a couple of extra spaces before the average and place a heading "AVERAGE" over the average column. Just guess at the spacing for this method.

Write an `int` method, `findLargestRow`, which takes as input a 2-dimensional array of `double` and returns the index of the row that has the largest sum.

Write an `int` method, `productDiagonal`, which has a square 2-dimensional array of integers as a parameter and returns the product of the main diagonal. The main diagonal refers to the entries where row == column. When you declare the variable that you will use for the product, initialize it to 1, not 0. Can you solve this with a single for loop?

Suppose you have a 2-dimensional 8 x 8 array, called `table`. Draw a little sketch of such a table below.

Suppose x and y are integers with values in the range [1,6]. Suppose you wished to refer to the four "neighbors" of `table[x][y]` (neighbors are above, below, to the left, and to the right).

The position directly above `table[x][y]` is `table`[_____] [_____].

The position directly below `table[x][y]` is `table`[_____][_____].

The position directly left of `table[x][y]` is `table`[_____][_____].

The position directly right of `table[x][y]` is `table`[_____][_____].

Write a method, `countIsolatedZeros`, which takes a 2-dimensional integer array as a parameter and returns the number of zeros that have no neighbors (above, below, left, or right) that are also zeros. It may be helpful to have a helper method that determines if indices are valid. For example, `indicesValid` could take as input a 2-dimensional integer array and two integer parameters for a row and column, and return true if the indices refer to a valid position in the table.

If you create an array of objects, you are really creating an array of references to objects. Each reference in the array is initialized to `null` by default. You can traverse the array with nested loops and create the objects themselves, assigning the array references to point to the objects. Can you guess what the code below is doing? Draw the first two rows of the chessboard array and enter the appropriate values.

```
Square[][] chessBoard = new Square[8][8];
String[] color = {"red","black"};
for(int i = 0; i < chessBoard.length; i++)
{
    for(int j = 0; j < chessboard[i].length; j++)
    {
        // assume Square constructor expects color
        chessboard[i][j] = new Square(color[(i+j)%2]);
    }
}
```

Big-Oh (Section 1.2)

Big-oh notation is used to measure of the efficiency of algorithms. It is not the only measure used, but it is the most common measure used. The big-oh of an algorithm is used to predict how the time (or space) of an algorithm grows as the size of the input data grows. It represents a growth rate.

We can calculate the big-oh of any mathematical function g(n), written O(f(n)), where f(n) refers to the largest order term in the function g(n). We do not specify the constant associated with the term.

Key ideas of big-oh:

1. The big-oh of the function: $4n^{13} - 2n^{10} + n^2$ is O(n^{13}).
2. If the biggest term is a constant (with no reference to the size n), the function is O(1), a constant function.
3. If the function makes use of two variables, we can specify the big-oh with two variables: 4m + 3n is O(m+n). Function: 7mn –m – 3n is O(mn).
4. The following are considered bad form for big-oh:
   a. Bad form: O($3n^2$). // Do not use the constant: O(n^2) is better.
   b. Bad form: O($\frac{n}{2}$). // Again, do not use the constant: O(n) is better.
   c. Bad form: O(n^2+n). // Only use the high order term: O(n^2) is better.
   d. Less bad form: O($\log_2 n$) // Better to use just O(log n). You will see the first form in some books. One log base is a constant times another log base, for bases greater than 1 (e.g., $\log_2 n$ is approximately 3.3219 * $\log_{10} n$).

Compute the big-oh of the following functions:

1. g(n) = $4n^3 - 3n^2 + 5n^4$ ________n^4__________
2. g(n) = 12 + $2\log_2 n$ ______logn_____________
3. g(n) = 4n + $n\log_3 n + 2^n$ _______nlogn___________

The following big-oh notations are not in the best form. Correct them.

1. O($3n^2$) _______n^2___________
2. O($\frac{n}{2}$) _______n___________
3. O(n^2 + n) ______n^2_____________
4. O($\log_2 n$) ________logn___________

Actually, big-oh is an upper bound. When we say that a function is O(n^3), for example, we are saying that the growth rate is less than or equal to some constant times n^3. So, $4n^{13}$ –

$2n^{10} + n^2$ is also $O(n^{14})$, or $O(n^{100})$ for that matter. But, the closer we are to getting the "real" high-order term, the more information we are giving. For example, suppose someone wanted to know how long it takes on average to get to Asheville from Boone by car. One could say:

1. "It takes less than 10 minutes."
2. "It takes less than 2 hours."
3. "It takes less than 8 hours."

Statements 2 and 3 are both correct, but statement 2 is more helpful. Statement 1 is WRONG, though. If you are not sure if a function has a high-order term of n^2 or n^3, you will be correct in saying $O(n^3)$, though perhaps you are not as informative as you could be. You might be wrong with $O(n^2)$. It is better to be less informative than to be wrong.

Suppose we have written an algorithm and implemented it in Java. Now we want to know how efficient our algorithm is. How fast is it going to run? This is a hard question. Some computers are faster than other computers. Some compilers translate to more efficient code than other compilers. We need a way to measure the efficiency of the algorithm, not the efficiency of the computer or the compiler. Big-oh is the way we do this.

Consider the following three functions. They are all O(______).

$T_1(n) = 4n^3 – 200n^2 + 5n +20$
$T_2(n) = 0.06n^3 + 2n^2 - n - 50$
$T_3(n) = 2n^3 + 104n^2 – 1000$

The three functions were evaluated for five values of n, and the table below shows the results. When a function is $O(n^3)$, it doesn't take very big values of n to get very large running times, whatever the constants might be.

Function	n=100	n=200	n=400	n=500	n=1000
$T_1(n)$	2,000,520	24,001,020	224,002,020	450,002,520	3,800,050,200
$T_2(n)$	75,850	559,750	4,159,550	7,999,450	61,998,950
$T_3(n)$	3,039,000	20,159,000	144,639,000	275,999,000	2,103,999,000

Consider the following ratios between a given function at one size and that same function at half that size. Look across the bottom row of ratios. Even though the exact functions (T_1, T_2, and T_3) are quite different, the fact that they all share n^3 as a high-order term means that their growth rates are similar as n gets larger and larger.

$T_1(200)/T_1(100) = 11.9$	$T_2(200)/T_2(100) = 7.38$	$T_3(200)/T_3(100) = 6.634$
$T_1(400)/T_1(200) = 9.3$	$T_2(400)/T_2(200) = 7.43$	$T_3(400)/T_3(200) = 7.17$
$T_1(1000)/T_1(500) = 8.4$	$T_2(1000)/T_2(500) = 7.75$	$T_3(1000)/T_3(500) = 7.6$

Recall that $T_1(n) = 4n^3 - 200n^2 + 5n + 20$. As n gets bigger, the lower order term do not affect the growth rate much. We can approximate $T_1(100{,}000)$ as $4 \times (100{,}000)^3$. We can approximate $T_1(200{,}000)$ as $4 \times (200{,}000)^3$. Notice what happens in the following ratio:

$$\frac{4(200{,}000)^3}{4(100{,}000)^3}$$

The constant 4 cancels. The ratio is 8. Notice that in the calculations of the ratios, as n got bigger all three of the $O(n^3)$ functions got closer and closer to 8 when the input size doubled.

Using Big-Oh to Predict Running Times

If we are given the big-oh of a function, and we know the function evaluation at one size, we can approximate the evaluation at a different size, even if we don't know the other terms or the constant. Suppose we are told that our function T has a big-oh of O(f(n)). We are told that the function evaluates to answer1 when n = size1. We want to predict the answer when n = size2. Here is a formula by which we can do that prediction.

$$answer2 \approx \frac{f(size2)}{f(size1)} \times answer1$$

Examples:

1. An algorithm is $O(n^2)$. The running time is 50 units when n = 10,000. Predict the running time when n = 30,000.

   Solution: $f(n) = n^2$, size1 = 10,000, size2 = 30,000, answer1 = 50. We solve for answer2:

   $$answer2 = \frac{(30{,}000)^2}{(10{,}000)^2} \times 50 = \frac{30{,}000 \times 30{,}000}{10{,}000 \times 10{,}000} \times 50 = 450$$

   FYI: Functions that are $O(n^2)$ are called quadratic functions. When you double the size of the input, the running time should go up 2^2 or 4 times. When you triple the size of the input, the running time should go up 3^2 or 9 times.

2. An algorithm is $O(n^3)$. The running time is 50 units when n = 10,000. Predict the running time when n = 30,000.

   Solution: $f(n) = n^3$, size1 = 10,000, size2 = 30,000, answer1 = 50. We solve for answer2:

   $$answer2 = \frac{(30{,}000)^3}{(10{,}000)^3} \times 50 = \frac{30{,}000 \times 30{,}000 \times 30{,}000}{10{,}000 \times 10{,}000 \times 10{,}000} \times 50 =$$

FYI: Functions that are $O(n^3)$ are called cubic functions. When you double the size of the input, the running time should go up 2^3 or 8 times. When you triple the size of the input, the running time should go up 3^3 or 27 times.

3. A function is O(n). The running time is 50 units on size 10,000. Predict the running time on input of size n = 30,000.

FYI: Functions that are O(n) are called linear functions. They seem to make the most sense. When the input size doubles, the running time doubles. When the input triples, the running time triples.

4. A function is O(log n). The running time is 50 units on size 10,000. Predict the running time on n = 30,000. The first time that you do the arithmetic, use $\log_{10}$ on your calculator. Then do the arithmetic with ln ($\log_e$) on your calculator. Finally, do it with $\log_2$. If your calculator does not do log base 2, you can use log 10. To determine $\log_2 x$, calculate $\log_{10} x / \log_{10} 2$.

$$answer2 = \frac{\log_{10} 30{,}000}{\log_{10} 10{,}000} \times 50 =$$

$$answer2 = \frac{\ln 30{,}000}{\ln 10{,}000} \times 50 =$$

$$answer2 = \frac{\log_2 30{,}000}{\log_2 10{,}000} \times 50 =$$

Now you see why the base of the logarithm doesn't matter. The ratios stay the same. Functions that are O(log n) are called logarithmic functions. They do not grow very fast at all! If you want to see the running time double, you need to pick an input size that is some constant to a power of n, and then pick that same constant to a power of 2n for the second size. For example, try input size 10,000 = 10^4, and input size 100,000,000 = 10^8.

5. A function is $O(2^n)$. The running time is 50 units on size 50. Predict the running time on size 100.

$$answer2 = \frac{2^{100}}{2^{50}} \times 50 = 2^{100-50} \times 50 =$$

Now predict the running time on an input just one larger, $n = 51$.

FYI: Functions that have the variable n in the exponent are called exponential functions. They grow *very* fast. Note that when you added just one element to the input, the running time doubles!

6. A function is $O(1)$. The running time is 50 units on input size 10,000. Predict the time on size 100,000.

$$answer2 = \frac{1}{1} \times 50 =$$

FYI: Functions with a constant as the high order term are called constant functions. It doesn't matter what the size of the input is. The running time stays the same.

Class Experiment

In the first lab, you used the algorithm labeled "Algorithm 2" below to shuffle a deck of cards. "Algorithm 1" below also correctly shuffles a deck of cards but its big-oh complexity is different. Here are the two algorithms:

Algorithm 1 Pseudocode:

```
1. One at a time, add the ints 0 through n-1, in order, to an ArrayList of Integer.
2. for (int i = 0; i < n-1; i++) {
3.     index1 = choose a random number between 0 and n-1.
4.     x = remove the Integer at position index1 in the ArrayList.
5.     index2 = choose a random number between 0 and n-1.
6.     add Integer x at position index2 in the list.
7. }
```

Algorithm 2 Pseudocode:

```
1. One at a time, add the ints 0 through n-1, in order, to an ArrayList of Integer.
2. for (int i = n-1; i > 0; i--) {
3.     index1 = choose a random number between 0 and i
4.     x = get Integer at position index1
5.     y = get Integer at position i
6.     set the Integer at position i to y
7.     set the Integer at position index1 to x
8. }
```

These algorithms are implemented on the student machine. Your instructor will show you where you can find them and will run them for you during class. The implementations will allow you to choose the data size for n and will report running times. Choose some running times and see if you can figure out the big-oh complexity of each of the algorithms. Note that if you choose a size that is too small, your running time will be reported as 0. Choose a larger size.

Algorithm 1

Size						
Run Time						

Big-Oh: _______________

Algorithm 2

Size						
Run Time						

Big-Oh: _______________

When you want to figure out the big-oh of an algorithm, you want to determine the high-order term of the number of steps the algorithm will take. Sometimes figuring out the big-oh can require lots of advanced mathematics, often involving probability. There are computer science journals that deal with the complexity of algorithms. There are algorithms for which the high-order term for the number of steps is not known, but researchers have been able to prove upper bounds. So the calculation of big-oh can be complicated.

For us, however, there are some rules of thumb that we can use that usually give us a good guess for the big-oh complexity of an algorithm. It is amazing how often this good guess is correct.

Rules of Thumb to Guess the Complexity:

1. Statements unaffected by the input size are constant. Examples: Simple assignment statements, comparisons and arithmetic.

2. Be careful: One line of code is not necessarily a constant. For example the line might call a method that is not constant; check out the complexity of the method.

3. Consecutive blocks of code: Add together the complexities of the blocks. The result is the same as the largest block.

4. `If` statements: Choose the largest complexity among the conditions (which is usually constant), and the largest complexities of all the possible `if/then/else` blocks. Note, here is one place we could overestimate. A more mathematical analysis might show that the worst case won't happen.

5. Loop: Multiply the number of iterations of the loop by the complexity inside the loop. `While` loops may be particularly hard to determine. Try to argue some upper bound if you are not sure how many iterations the loop will have.

6. Be aware: It is easy to overestimate a loop, especially if there is an if statement inside the loop and, by multiplying, we are counting the worst case of the `if` statement at each iteration.

7. Nested loops: Multiply the big-oh of the code inside the loops by the numbers of iterations of each nested loop. Again this may overestimate, especially if we repeatedly are too high on the number of iterations.

See how well you do at figuring out the big-oh of the six algorithms on the next page.

```
public int alg1 (int n)
{
     int steps = 0;
     for (int i = 0; i < n; i++)
     {
         steps++;
     }
     return steps;
}
```

```
public int alg2 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < n*n; j++)
        {
            steps++;
        }
    }
    return steps;
}
```

```
public int alg3 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < i; j++)
        {
            steps++;
        }
    }
    return steps;
}
```

```
public int alg4 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            steps++;
        }
    }
    return steps;
}
```

```
public int alg5 (int n)
{
    int steps = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j =0; j < i*i; j++)
        {
            for (int k = 0; k < j; k++)
            {
                steps++;
            }
        }
    }
    return steps;
}
```

```
public int alg6 (int n)
{
   int steps = 0;
   for (int i = 0; i < n; i++)
   {
       for (int j =0; j < i*i; j++)
       {
           steps++;
           if (j % i == 0)
           {
              for (int k = 0; k < j; k++)
              {
                  steps++;
              }
           }
       }
   }
   return steps;
}
```

Exceptions (Appendix C)

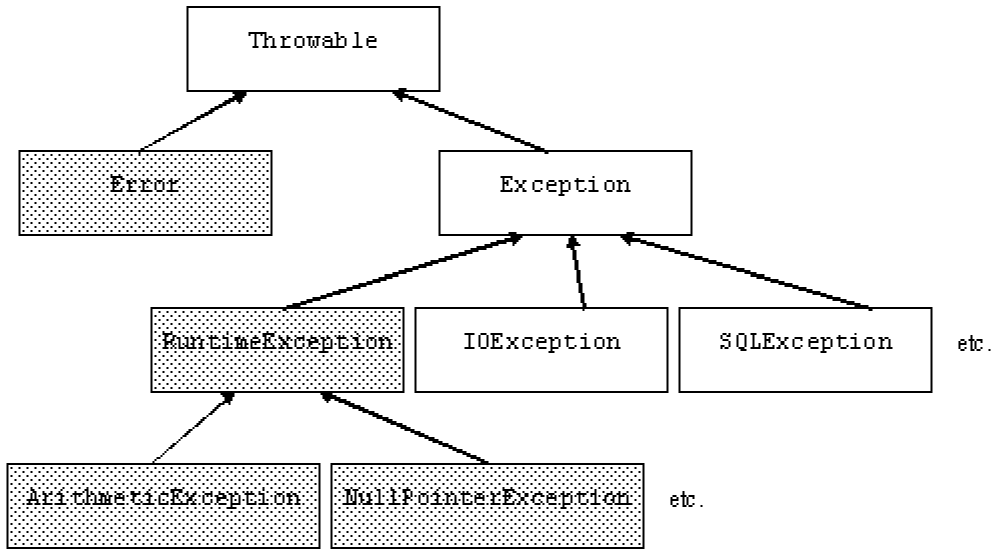
In section 2.1 of our textbook, the author tells us that the goal of chapter 2 "is to be able to write general-purpose classes that can be used by many different programs." Most of the material in chapter 2 was covered in CS 1, but exceptions were not. The rest of this handout deals with exceptions and how they are used in general-purpose classes.

When you write applications programs for yourself, you generally know how you want to treat bad input data. For example, if the user enters bad data, perhaps you print an error message and allow him or her to enter the data again.

When you write classes that are used by other programmers and your code encounters errors in the data it receives, it is better to tell the outside code that an error occurred and let the other program decide exactly how to handle the error. Java exceptions provide the mechanism for doing this. Your code notifies the outside code that an error occurred by *throwing* an exception. The programmer who writes the outside code needs to know that your code might throw an exception so he or she can design the outside code to properly *catch* any exceptions that are thrown.

We will first examine how you can throw exceptions from your code, and then how you should write any code that calls on other classes that might throw exceptions. First, let's see what exceptions actually are.

`Exception` is a subclass of a class called `Throwable`. `Exception` has many subclasses of its own. One particular subclass is called `RuntimeException` and it has many of its own subclasses. `Throwable` objects are categorized into two groups: (1) `Errors`, `RuntimeExceptions`, and subclasses of `RuntimeExceptions` (these are called "unchecked" exceptions and are gray in the picture), and (2) `Exceptions` and subclasses of `Exception` other than `RuntimeException` (these are called "checked" exceptions). We will come back to this distinction later in this section.



Throwing Exceptions

Take a look at one of the constructors in the `ArrayList` class. `ArrayList` is a library class that is used by thousands of programmers around the world. When a programmer declares a new `ArrayList`, he or she can specify the size of the list that is desired. Of course a negative size makes no sense. If the constructor detects that a negative size has been requested, it throws an `IllegalArgumentException`. `IllegalArgumentException` is one of the subclasses of `RuntimeException`.

```
127     public ArrayList(int initialCapacity) {
128         super();
129         if (initialCapacity < 0)
130             throw new IllegalArgumentException("Illegal Capacity: "+
131                                                initialCapacity);
132         this.elementData = new Object[initialCapacity];
133     }
```

On line 129 the constructor checks to see if its parameter is negative, and if it is then a new object of type `IllegalArgumentException` is created. The constructors of `Exception` classes can take a string as a parameter. This string will be printed to the output if the exception is thrown. Such strings help other programmers understand what went wrong.

When we get to chapter 3, we will write a class called `DoubleArraySeq`. This is a container class for doubles and it keeps the doubles in an array. Here is one of the constructors for that class:

```
/**
 * Initializes an empty sequence with the specified initial capacity.
 *
 * @postcondition This sequence is empty and has an initial capacity of
 *                initialCapacity.
 *
 * @param initialCapacity
 *            initial size of the array
 * @throws OutOfMemoryError
 *             if there is insufficient memory for: new
 *             double[initialCapacity].
 * @throws IllegalArgumentException
 *             if initialCapacity is negative
 */
public DoubleArraySeq(int initialCapacity)
    throws OutOfMemoryError, IllegalArgumentException
{
    // TODO
    if (initialCapacity < 0)
    {
        throw new IllegalArgumentException(
            "initialCapacity is negative: " + initialCapacity);
    }
    data = new double[initialCapacity];
    manyItems = 0;
    currentIndex = 0;
}
```

Notice that the constructor checks its parameter and throws an `IllegalArgument-Exception` if the parameter is negative. Also notice that in the Javadoc comment above the constructor, it is pointed out that the constructor might also throw an `OutOfMemoryError`. Any time that you request space for an object by using the `new` keyword, it is possible to have no more space in RAM and to have Java throw this `Error`. Look back at the hierarchy of `Throwable` objects and you will see that `Error` is a subclass of `Throwable`. Most of the time programmers don't explicitly point out that their methods might encounter the `OutOfMemoryError`, but our author makes all such possibilities very explicit in his code because he is trying to educate us.

There is another difference between the constructor in `ArrayList` and the constructor in our author's `DoubleArraySeq` class, and that is the use of the word `throws` after the parameter list of the constructor. If a constructor or method throws a checked exception, the compiler insists that we make that possibility known by using a `throws` clause. If we do not declare our checked exceptions in this way, the code will not compile. It is not mandatory, however, to have this `throws` clause in the `DoubleArraySeq` constructor because one of the possible objects being thrown is an `Error` and the other is a subclass of `RuntimeException` (an unchecked exception). It is not wrong to use a `throws` clause for unchecked exceptions and errors – just not required. The reason we say that some exceptions are "checked" is because the compiler mandates how we are to deal with them and checks to be sure that we do. It will not compile our code if we break the rules. There are other rules for managing checked exceptions as we shall see.

For practice, write a statement that throws an `IndexOutOfBoundsException` if the `int index` has a value that is not in the range `[0, numArray.length -1]`. Make the message of the exception be "Invalid index: " followed by the value of the variable, `index`.

Making Up New Exception Types

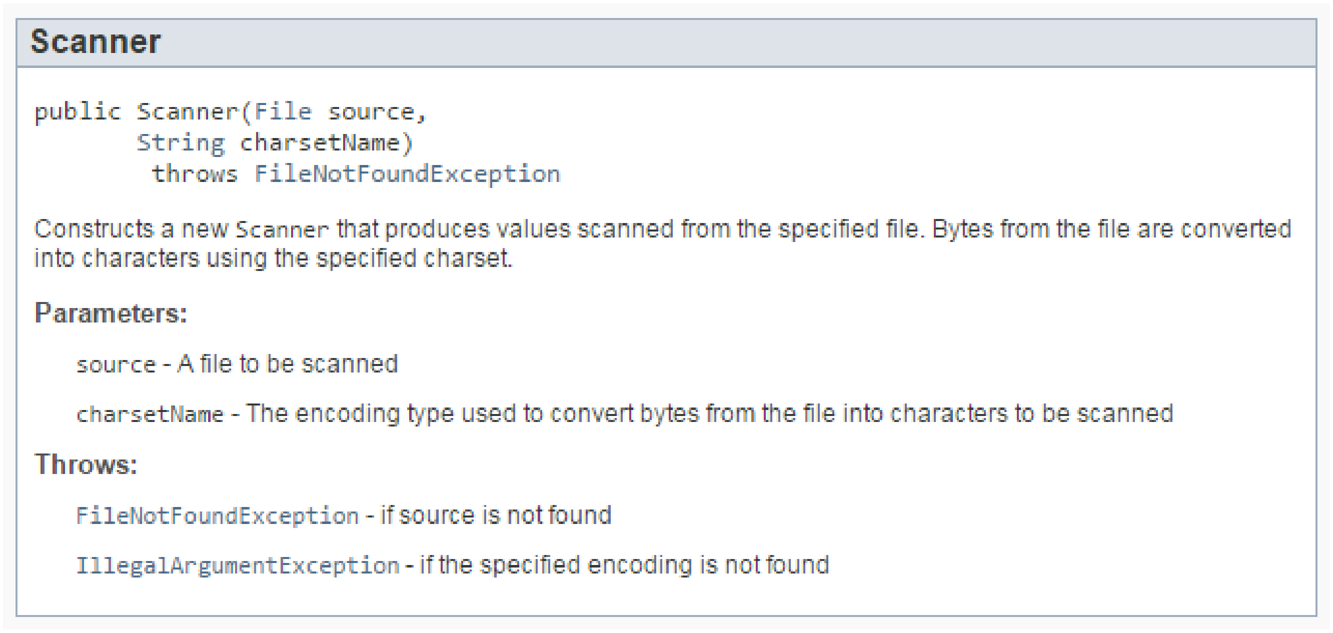
Note that you can make up your own `Exception` classes if you want, although this need is rare. You can extend `Exception` if you want to make your new exception type be checked by the compiler, or you can extend `RuntimeException` if you want to create an unchecked exception type. You don't need any methods in your new class, only two constructors. You need a default constructor and one that takes a `String` as a parameter. Each constructor simply calls the superconstructor. The one with the `String` parameter passes that `String` on up to the superconstructor.

Here is an example of a new exception called `DoesNotFollowDirectionsException`:

```
public class DoesNotFollowDirectionsException extends RuntimeException
{
    public DoesNotFollowDirectionsException() {
        super();
    }

    public DoesNotFollowDirectionsException(String message) {
        super(message);
    }
}
```

Handling Exceptions

Now we need to discuss how we should write code that calls methods that might throw exceptions. In order to know whether a method might throw an exception or not, you have to look at the documentation for that method. The Java API for the `Scanner` class includes the following documentation for one of the constructors:



Scanner

```
public Scanner(File source,
               String charsetName)
                throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - A file to be scanned

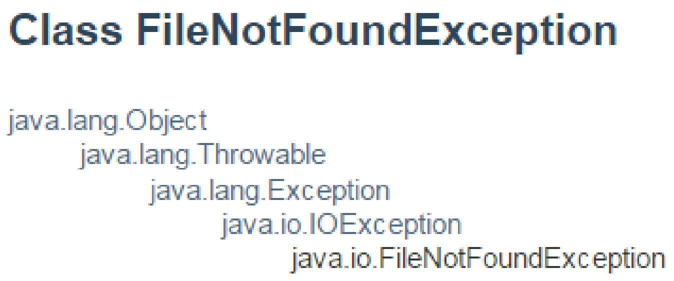
`charsetName` - The encoding type used to convert bytes from the file into characters to be scanned

Throws:

`FileNotFoundException` - if source is not found

`IllegalArgumentException` - if the specified encoding is not found

We can see that the constructor can throw both a `FileNotFoundException` and an `IllegalArgumentException`. Now we need to know whether those exceptions are checked or unchecked. So we look each of them up in the API. At the top of the `FileNotFoundException` page we see this:

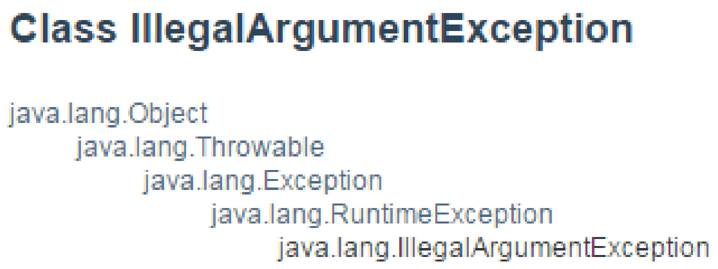


Class FileNotFoundException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

Looking at the inheritance hierarchy we see that `FileNotFoundException` is a subclass of `IOException`, which is a subclass of `Exception`. So `FileNotFoundException` is a checked exception.

At the top of the `IllegalArgumentException` page we see this:



Class IllegalArgumentException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

Here we note that `IllegalArgumentException` is a subclass of `RuntimeException` and is, thus, unchecked.

It is not bad practice to deal with both checked and unchecked exceptions in the same way, but the rules described below are mandatory for checked exceptions. Before we get to those rules, note that if you do not deal with an unchecked exception in your code, and that exception gets thrown at runtime, your program will crash. An error message will print to the console that shows that the exception occurred.

There are two ways to deal with a checked exception that might get thrown by a method that your code calls: (1) put the risky code into a `try` block, and put a `catch` block after the `try` block that will catch and deal with the exception if it gets thrown, or (2) mark the method that contains the risky code with a `throws` clause (after the parameter list) and let any thrown exception propagate to your method's caller. You should do the first if you know how to deal with the exception, and you should do the second if you don't. You can actually write code that does both of these things. You can put the risky code into a `try` block followed by a `catch` block that partially deals with the exception and then does its own `throw` to send the exception on up the runtime chain of method calls.

Try and Catch Blocks

If a method is going to deal with a potential exception internally, the line of code that could generate the exception is placed inside a `try` block. There might be other code inside the `try` block, before and/or after the risky line(s). Any code that depends upon the risky code's success should be in the `try` block, since it will automatically be skipped if the exception occurs.

```
    try
    {
        risky code and code that depends on the risky code succeeding
    }
```
There is usually at least one `catch` block immediately after the `try` block. A `catch` block must specify what type of exception it will catch.

```
catch (ExceptionClassName exceptionObjectName)
{
    code using methods from ExceptionClassName
}
```

There can be more than one catch block, each one marked for a specific exception class. The exception class that is caught can be any class in the exception hierarchy, either a general (base) class, or a very specific (derived) class. You should put the more specific catch blocks first since the first applicable catch block will be the one that gets used and the others will be skipped. The catch block(s) must handle all checked exceptions that the try block is known to throw unless you want to let the exception propagate back to the method that called your method.

It is possible to have a try block without any catch blocks if you have a finally block, but any checked exceptions still need to be caught, or the method needs to declare that it throws them. We will cover finally later in this section.

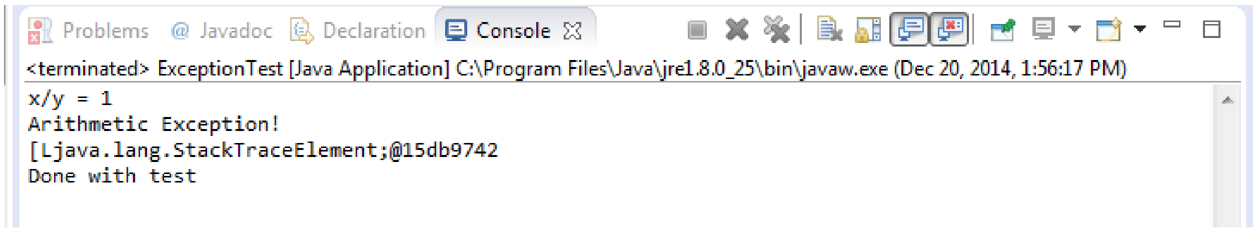
If an exception occurs within a try block, execution jumps to the first catch block whose exception class matches the exception that occurred (Java uses an instanceof test). Any steps remaining in the try block are skipped. If no exception occurs, then the catch blocks are skipped. The catch blocks will also be skipped if an exception that is not caught occurs, such as a RuntimeException, or an exception that the method declared it throws.

If you declare a variable within a try block, it will not exist outside the try block, since the curly braces define the scope of the variable. You will often need that variable later, if nowhere else other than the catch or finally blocks, so you would need to declare the variable before the try.

If you declare but don't initialize a variable before a try block, and the only place you set a value for that variable is in the try block, then it is possible when execution leaves the try/catch structure that the variable never received a value. So, you would get a "possibly uninitialized value" error message from the compiler, since it actually keeps track of that sort of thing. Usually this happens with object references; you would generally initialize them to null.

The following program will print the first result, and then fail while performing the division for the second equation. Execution will jump to the catch block to print our message on the screen. Note: ArithmeticException is one of the exceptions that you are not required to catch, but you can still catch it if you wish.

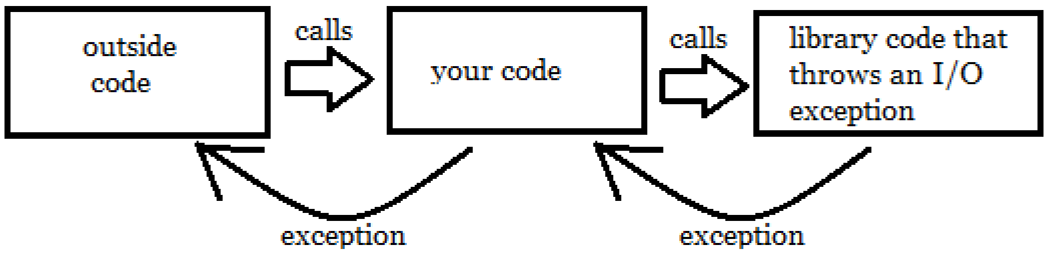
```
public class ExceptionTest {
  public static void main(String[] args) {
    int i, j, x = 5, y = 5, z = 0;
    try {
      i = x/y;
      System.out.println("x/y = " + i);
      j = x/z;
      System.out.println("x/z = " + j);
    }
    catch(ArithmeticException e) {
      System.out.println("Arithmetic Exception!");
      System.out.println(e.getStackTrace());
    }
    System.out.println("Done with test");
  }
}
```



```
<terminated> ExceptionTest [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (Dec 20, 2014, 1:56:17 PM)
x/y = 1
Arithmetic Exception!
[Ljava.lang.StackTraceElement;@15db9742
Done with test
```

File I/O Exceptions

Most methods in the I/O classes throw `IOException`, which is a checked exception that you are required to handle. Much of the time, however, your code will not be able to deal with a file I/O exception locally and you will be better off letting the exception propagate (just using a `throws` declaration). For example, if you are writing code that relies on something like a file name being passed to it from outside your class and your code receives an invalid file name, you can't know what the correct file name should have been. So you pass the exception on to your caller and let that caller catch it and correct the error.



Using Multiple Catch Blocks

It is possible that a statement might throw more than one kind of exception. You can list a sequence of catch blocks, one for each possible exception. Remember that there is an object hierarchy for exceptions. Since the first one that matches is used and the others skipped, you should put a subclass class first and its parent class later. You will actually get a compiler error if you list a parent class before a child class, as you have "unreachable code." In this example, the code in the `try` block could throw `NumberFormatException` during the parsing, and `ArithmeticException` while doing the division, so we have catch blocks for those specific cases. The more generic `Exception` would catch other problems, although in this case it isn't possible to cause any other type of exception.

```java
import javax.swing.JOptionPane;

public class MultiCatchTest {

    public static void main(String[] args)
    {
        int n1, n2;
        try
        {
            n1 = Integer.parseInt(JOptionPane.showInputDialog("Enter a number"));
            n2 = Integer.parseInt(JOptionPane.showInputDialog("Enter a number"));
            System.out.println(n1 + " / " + n2 + " = " + n1/n2);
        }
        catch (NumberFormatException e)
        {
            System.out.println("Number Format Exception occurred");
        }
        catch (ArithmeticException e)
        {
            System.out.println("Divide by Zero Exception occurred");
        }
        catch (Exception e)
        {
            System.out.println("General Exception occurred");
        }
    }
}
```

Guaranteeing Execution of Code – The Finally Block

To guarantee that a chunk of code runs, whether or not an exception occurs, use a `finally` block after the `try/catch` blocks. There may be only one `finally` block and it must be last. The code in the `finally` block will *almost* always execute. If an exception causes a `catch` block to execute, the `finally` block will be executed after the `catch` block. If an uncaught exception occurs, the `finally` block executes and then execution exits the method and the exception is thrown to the method that called this method. If either the `try` block or a `catch` block executes a `return` statement, the `finally` block executes before control leaves the method. If either the `try` block or a `catch` block calls `System.exit`, the `finally` block will not execute. If a `finally` block executes a `return` while an uncaught exception is pending, the exception is stifled; that is, it just disappears.

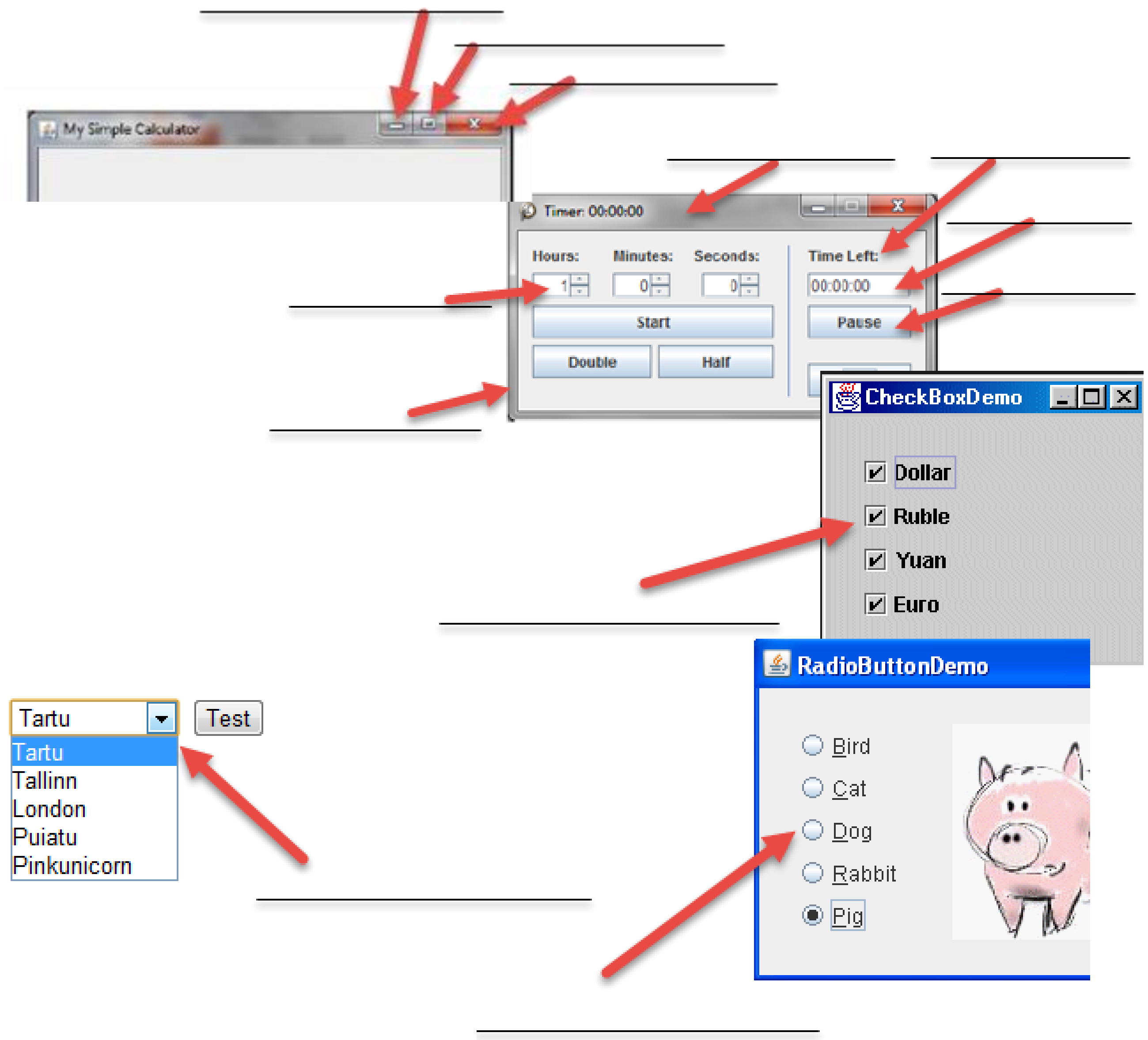
Here is an interesting piece of code to run. It illustrates much of what we have discussed in this section. We will run it during class.

```java
import javax.swing.JOptionPane;

public class FinallyTest {
  public static void main(String[] args) {
    System.out.println("Returned value is " + go());
  }

  public static int go() {
    int choice = 0;
    try {
      String name = JOptionPane.showInputDialog("Enter your name: ");
      System.out.println("MENU:");
      System.out.println("1 - normal execution");
      System.out.println("2 - uncaught ArithmeticException");
      System.out.println("3 - return from try block");
      System.out.println("4 - call System.exit");
      System.out.println(
            "5 - return 5 from finally after ArithmeticException");
      System.out.println(
            "6 - return 6 from finally after try returns -1");
      System.out.println("X - catch NumberFormatException");
      choice = Integer.parseInt(JOptionPane.showInputDialog("Enter your choice: "));

      if (choice == 1) System.out.println("Hello " + name);
      else if (choice == 2) System.out.println("1 / 0 =  " + 1/0);
      else if (choice == 3) return 3;
      else if (choice == 4) System.exit(1);
      else if (choice == 5) System.out.println("1 / 0 =  " + 1/0);
      else if (choice == 6) return -1;
    }
    catch (NumberFormatException e) {
      System.out.println("Number Format Exception occurred");
    }
    finally {
      System.out.println("Goodbye from finally block");
      if (choice == 5) return 5;
      if (choice == 6) return 6;
    }
    return 0;
  }
}
```

Graphical User Interfaces

Unless you make graphical user interfaces (GUIs) frequently, you will probably have to look up some of the details of their construction every time you make one.   Concentrate on remembering the big concepts of building GUIs and don't try to memorize every detail.

Let's label some parts so that we have common terminology.



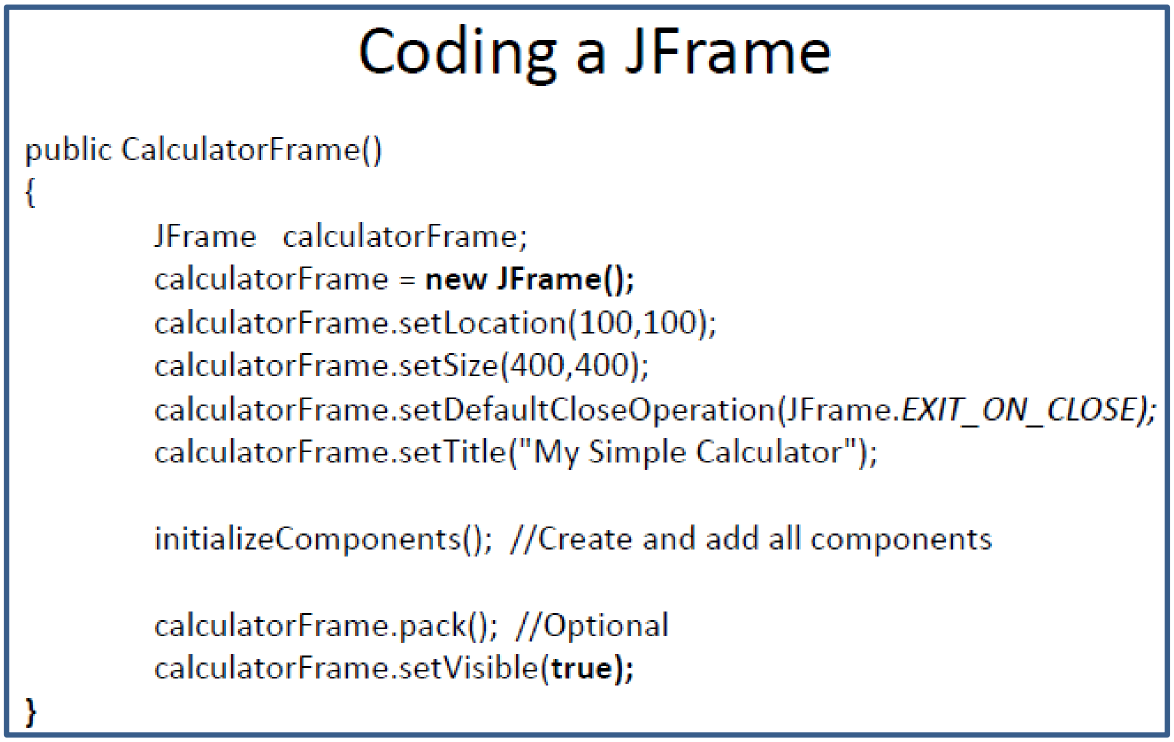
Swing

Swing is a toolkit used for making graphical user interfaces. It is part of Oracle's Java Foundation Classes (JFC), which is an API for making GUIs. Swing was developed to take the place of the Abstract Window Toolkit (AWT), which was platform-specific. Swing is platform-independent. The Swing classes and components are contained in the `javax.swing` package.

General Idea

The general idea of building a GUI with Swing is to do the following:

1. Declare and allocate a JFrame. This is the window. Set the properties you want your window to have.



Coding a JFrame

```
public CalculatorFrame()
{
        JFrame   calculatorFrame;
        calculatorFrame = new JFrame();
        calculatorFrame.setLocation(100,100);
        calculatorFrame.setSize(400,400);
        calculatorFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        calculatorFrame.setTitle("My Simple Calculator");

        initializeComponents();  //Create and add all components

        calculatorFrame.pack();  //Optional
        calculatorFrame.setVisible(true);
}
```

2. Organize the components of your GUI into categories. For each category, declare and allocate a JPanel. Decide on a layout manager for each JPanel and use it to add the components to the JPanel.

3. Decide on a layout manager for the JFrame and use it to add the JPanels to the JFrame.

```
JFrame testFrame = new JFrame()
JPanel  buttonPanel = new JPanel();
JButton jbTest = new JButton("Test Me");


buttonPanel.add(jbTest);
testFrame.add(buttonPanel, BorderLayout.PAGE_END);
```

4. Create and add listeners to the components that require them.

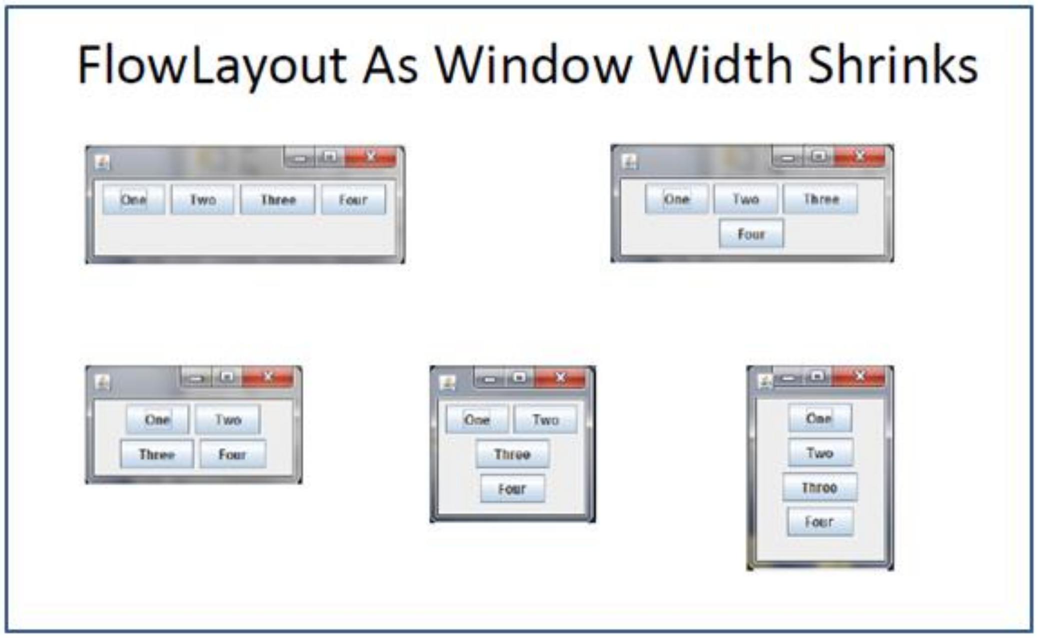
Java Tutorials

If you want to learn a lot about building GUIs (and I encourage this!), study the Java Tutorials online at http://docs.oracle.com/javase/tutorial/uiswing/index.html.

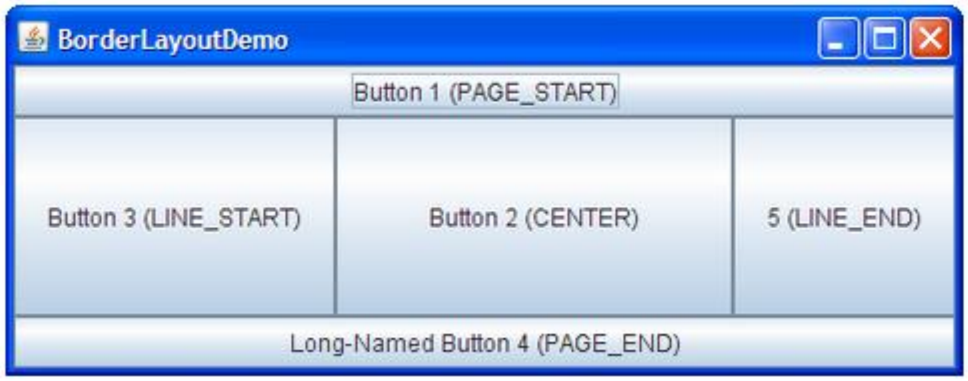
Layout Managers

There are numerous layout managers in JavaFX. Study them in the tutorials. Here are brief descriptions of a few.

- FlowLayout: This is the default layout manager for JPanels. It places components left to right and centers them within the panel. It allows them to move (flow) when the user resizes the window.


FlowLayout As Window Width Shrinks

- BorderLayout: The window is divided into five areas. You can add components to any of the five parts.



- BoxLayout: This is a more flexible version of FlowLayout.
- CardLayout: This is a more complicated layout that allows two or more components to share the same display space. It allows the user to choose between the components using a menu.

Events

Swing components can fire events. Buttons fire events when they get clicked by a user, for example. In order to have your GUI detect when a component has fired an event, you need to add an event listener to the component.

Java Listeners

There are many listeners in the Java API. Each of them is an interface with a small number of methods that need to be implemented in order to implement the interface. The interface we will use for the GUIs we build in 2440 is called ActionListener.

ActionListener

The ActionListener interface specifies that you implement one method called actionPerformed with this signature:

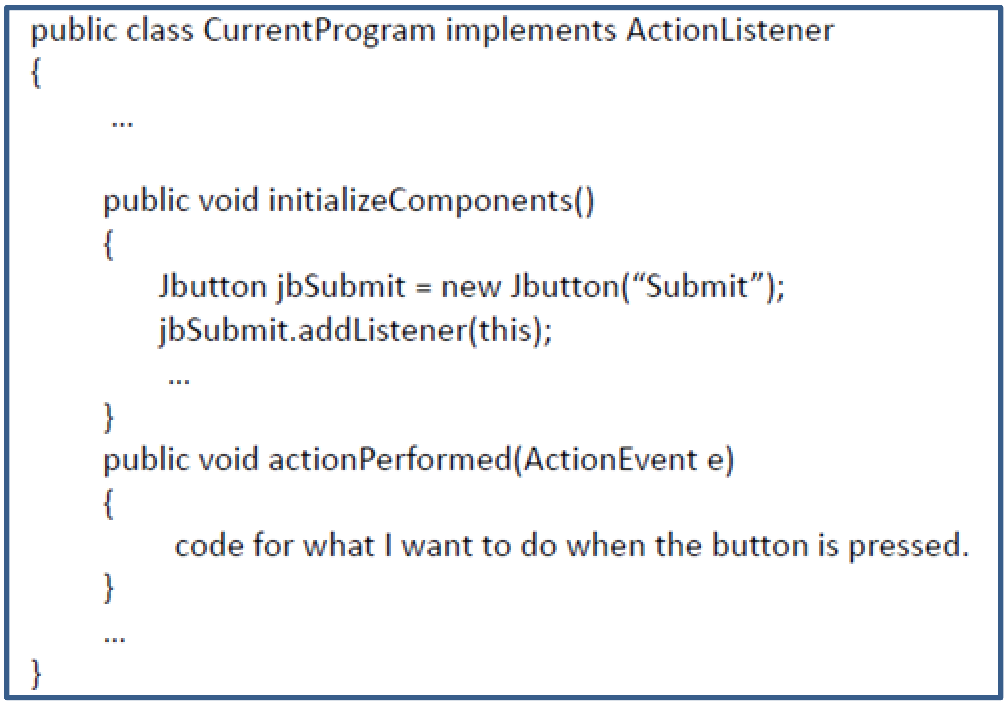
```
public void actionPerformed(ActionEvent e)
{
    ...//code that reacts to the action...
}
```

We will use a button as the component we want to listen to, since this is what the GUI you build in lab will need to do. In order to give a button the functionality you want it to have, you have to create a new Java class that implements the ActionListener interface. You then have to create an object of this new type and add it to the button using the button's addActionListener method.

There are three ways to do this, and we will discuss each of them below.

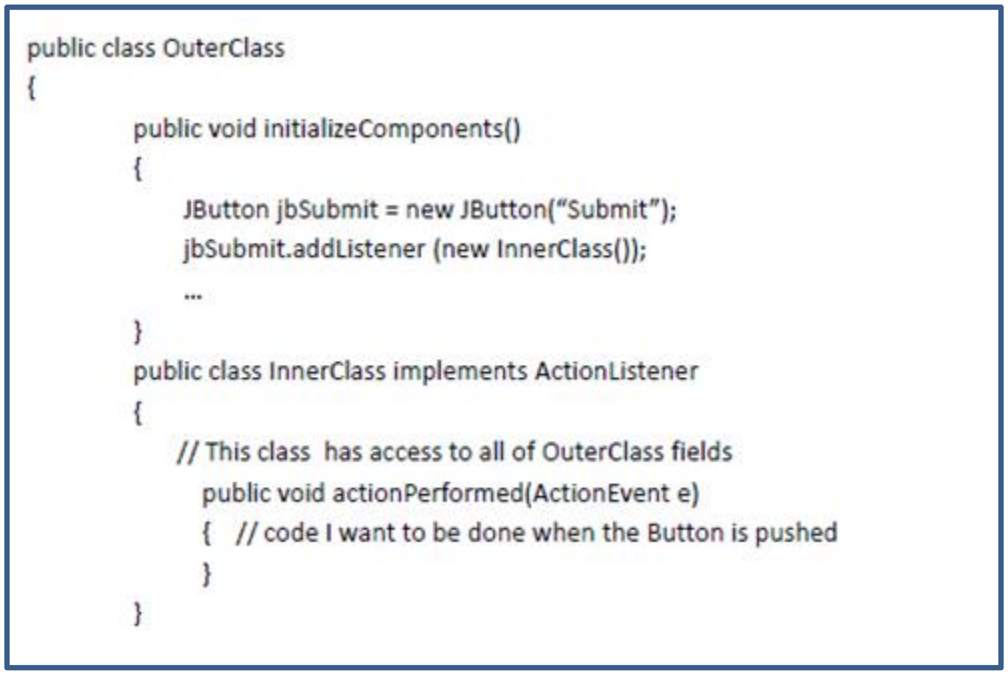
1. Have the GUI class itself implement ActionListener.
2. Put an inner class inside your GUI class that implements ActionListener.
3. Use an anonymous inner class.

GUI Class Implements ActionListener



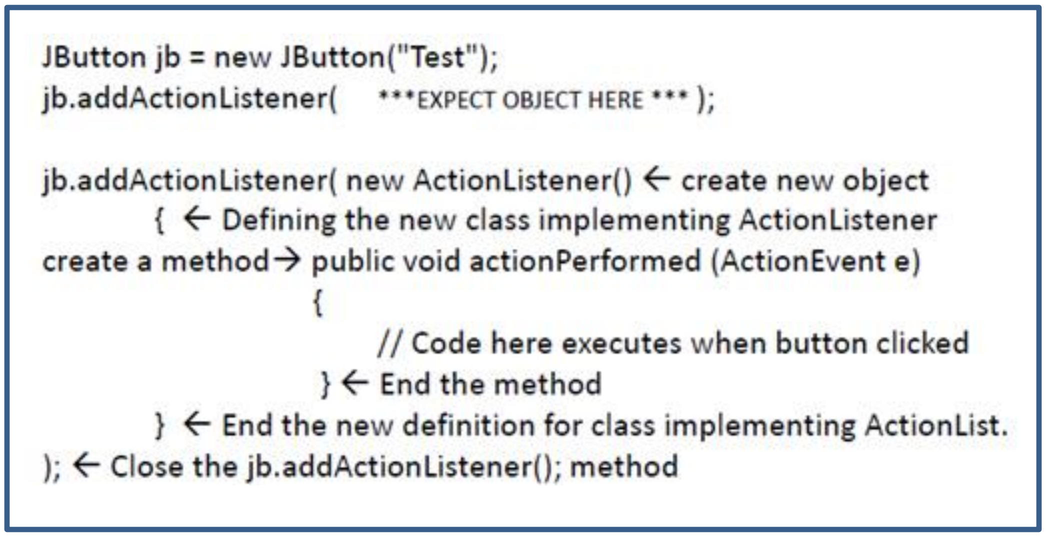
```
public class CurrentProgram implements ActionListener
{
         ...

        public void initializeComponents()
        {
              Jbutton jbSubmit = new Jbutton("Submit");
              jbSubmit.addListener(this);
               ...
        }
        public void actionPerformed(ActionEvent e)
        {
                code for what I want to do when the button is pressed.
        }
        ...
}
```

Inner Class



```
public class OuterClass
{
            public void initializeComponents()
            {
                  JButton jbSubmit = new JButton("Submit");
                  jbSubmit.addListener (new InnerClass());
                  ...
            }
            public class InnerClass implements ActionListener
            {
                  // This class  has access to all of OuterClass fields
                     public void actionPerformed(ActionEvent e)
                     {   // code I want to be done when the Button is pushed
                     }
            }
```

Anonymous Inner Class

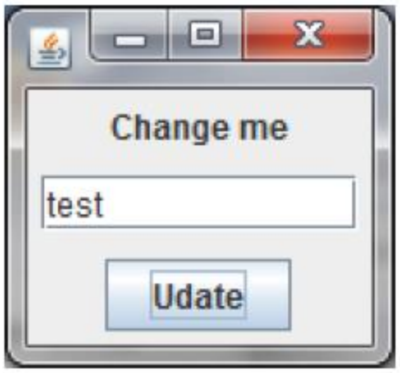


```
JButton jb = new JButton("Test");
jb.addActionListener(     ***EXPECT OBJECT HERE *** );

jb.addActionListener( new ActionListener() ← create new object
           { ← Defining the new class implementing ActionListener
create a method→ public void actionPerformed (ActionEvent e)
                           {
                                 // Code here executes when button clicked
                           } ← End the method
           } ← End the new definition for class implementing ActionList.
); ← Close the jb.addActionListener(); method
```

Example (uses anonymous class)

- Create a JFrame with a single label, text field, and button.
- When the button is pressed the label text is changed to the same text as is currently in the text field.
- If the text field is empty, do not update the label.

Example



```
public class TestFrame {

   private JFrame testFrame;
   private JLabel jlChangeMe;
   private JTextField jtfNewText;
    // constructor
    public TestFrame()
       {

          testFrame = new JFrame();
          testFrame.setLocation(100,100);
          testFrame.setSize(400,400);
          testFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          testFrame.setTitle("My Simple Label Changer");

          initializeComponents();

          testFrame.pack();
          testFrame.setVisible(true);
       }
       public void initializeComponents()
       {
          JButton jbUpdate = new JButton("Udate");
          jtfNewText = new JTextField(10);
          jlChangeMe = new JLabel("Change me");

          //Create the north panel
          // add the changeMe label from above to the panel
          //add the panel to the north section of the frame
          JPanel jpNorth = new JPanel();
          jpNorth.add(jlChangeMe);
          testFrame.add(jpNorth, BorderLayout.PAGE_START);
```

```
//Create the center panel , add the text field to it, and
 //add the panel to the center section of the frame
 JPanel jpCenter = new JPanel();
 jpCenter.add(jtfNewText);
 testFrame.add(jpCenter, BorderLayout.CENTER);

 //Create the south panel, add the button to it, and
 //add the panel to the south section of the frame
 JPanel jpSouth = new JPanel();
 jpSouth.add(jbUpdate);
 testFrame.add(jpSouth, BorderLayout.PAGE_END);

 // create the anonymous class with the required actionPerformed method.
 // when the button is clicked, the method updateClicked() will be called.
 jbUpdate.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e){
       updateClicked();
    }
 });

// this is what we do when the button is clicked
 public void updateClicked()
   {
     //Get the user text from the text field
     String t = jtfNewText.getText();

     //Check to see if it is blank
     if (!t.trim().equals(""))
     {
        //If it is not blank set the test on the label
        jlChangeMe.setText(t);
     }

   }
```

Don’t forget import Statements

- GUI Components: Importing from javax.swing:
  - import javax.swing.JButton;
  - import javax.swing.JFrame;
  - import javax.swing.JLabel;
  - import javax.swing.JPanel;
  - import javax.swing.JTextField;
  - Import javax.swing.*;   // lazy way
- Layouts and Listeners: Importing from java.awt
  - import java.awt.event.ActionListener;
  - import java.awt.BorderLayout;
  - Import java.awt.*;   // lazy again

Tips

- Your JFrame object should be a field.
- Any component you want to change or query in the program should be a field.
  - TextFields almost always a field.
  - JLabel make a field only if you want change the text.
  - JButton make a field only if you want to change the text written on the button.
  - Make all other components local variables.
- Use pack() to shrink the window down to its minimum size just before making visible.
- Add components to JPanels and add JPanels to your frame.

Tips

- initializeComponents() should call other methods to create the JFrame in pieces instead of being one big method.
- Have the method inside the listener class call some other method that does the work .
- This simple exercise used everything as a single Class. Your complicated projects should strive to separate the "Display" GUI classes from the "Data" classes (for an example, a GUI for a Battleship game should be a separate class)

Look at Some Components, Layouts, and Event Handling with Listeners

See if you can figure out what each example is doing.

The Object Class and its Methods (Sections 2.3, 2.4)

Most of this section comes from docs.oracle.com/tutorial/java/IandI/objectclass.html.

Object as a Superclass

The `Object` class, in the `java.lang` package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the `Object` class. Every class you use or write inherits the instance methods of `Object`. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from `Object` that are discussed in this section are:

- `protected Object` **`clone`**`() throws CloneNotSupportedException`
  Creates and returns a copy of this object.
- `public boolean` **`equals`**`(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `protected void` **`finalize`**`() throws Throwable`
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class` **`getClass`**`()`
  Returns the runtime class of an object.
- `public int` **`hashCode`**`()`
  Returns a hash code value for the object.
- `public String` **`toString`**`()`
  Returns a string representation of the object.

Note: There are some subtle aspects to a number of these methods, especially the `clone` method.

clone

If a class, or one of its superclasses, implements the `Cloneable` interface, you can use the `clone` method to create a copy of an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

Object's implementation of this method checks to see whether the object on which `clone` was invoked implements the `Cloneable` interface. If the object does not, the method throws a `CloneNotSupportedException`. If you write a `clone` method to override the one in Object, it must be declared as

```
protected Object clone() throws CloneNotSupportedException
```
or:
```
public Object clone() throws CloneNotSupportedException
```

If an object on which `clone` is invoked does implement the `Cloneable` interface, Object's implementation of the `clone` method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add "`implements Cloneable`" to your class's declaration. Then your objects can invoke the `clone` method in the Object class.

For some classes, the default behavior of `Object`'s `clone` method works just fine. If, however, an object contains a reference to an external object, say `ObjExternal`, you may need to override `clone` to get correct behavior. Otherwise, a change in `ObjExternal` made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override `clone` so that it clones the object and `ObjExternal`. Then the original object references `ObjExternal` and the clone references a clone of `ObjExternal`, so that the object and its clone are truly independent.

The following code is taken from our textbook's `Location` class. The `clone` method here is an override of the `clone` method in `Object`. Our author declares a new `Location` reference called `answer`. Then in a try block he attempts to use the `clone` method in `Object` to make a clone of the activating `Location` object. If that fails, it's because the "`implements Cloneable`" clause was forgotten at the top of the `Location` class. This seems a bit silly since it's likely that the same person writing the clone method also wrote the rest of the `Location` class, but on large software projects this makes sense. The missing clause would be discovered if some programmer on the project attempted to make a clone of a `Location` object.

```
public Location clone( )
{  // Clone a Location object.
    Location answer;

    try
    {
        answer = (Location) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {  // This exception should not occur. But if it does, it would probably
        // indicate a programming error that made super.clone unavailable.
        // The most common error would be forgetting the "Implements Cloneable"
        // clause at the start of this class.
        throw new RuntimeException
        ("This class does not implement Cloneable.");
    }

    return answer;
}
```

Note that the usual behavior of the above method is to use the `clone` method in `Object` to create a copy of the activating `Location` object and return it to the caller. For any class with fields that are not references to external objects, this type of `clone` method is sufficient.

In the next chapter we begin to implement container classes – classes similar to `ArrayList`, in that they are a container used to store many separate values. The container itself, in our first examples, will be an array. The container class will have a field that is a reference to the array. The array, however, is an external object. This means that we have to do something different when we clone a container class.

Look at the `clone` method here and circle the line that is different from the lines in the above clone method from the `Location` class.

```
public IntArrayBag clone()
{
    // Clone an IntArrayBag object.
    IntArrayBag answer;
    try
    {
        answer = (IntArrayBag) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {
        // This exception should not occur. But if it does,
        // it would probably indicate a programming error
        // that made super.clone unavailable. The most common
        // error would be forgetting the “Implements Cloneable”
        // clause at the start of this class.
        throw new RuntimeException
        ("This class does not implement Cloneable.");
    }
    answer.data = data.clone( );
    return answer;
}
```

That line of code is doing what we call a "deep copy." Since the `IntArrayBag` class contains an array reference as a field, when you want to clone an `IntArrayBag` object, you have to clone the array separately. The call to `super.clone()` merely copies the fields, not the external objects.

Draw a picture of a shallow copy and a deep copy of an object that contains external objects as fields.

equals

The `equals` method compares two objects for equality and returns true if they are equal. The `equals` method provided in the `Object` class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this operator gives the correct result. For objects, however, it does not. The `equals` method provided by `Object` tests whether the object references are equal—that is, whether the objects being compared are *the exact same object*.

To test whether two objects are equal in the sense of *equivalency* (whether they contain the same information in their fields or not), you must override the `equals` method. Here is an example of a `Book` class that overrides `equals`. Note that the programmer decided that two `Books` are equal if their ISBN numbers are the same.

```
public class Book {

    private String ISBN;
    // rest of code

    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals(((Book)obj).getISBN());
        else
            return false;
    }
}
```

Consider this code that tests two instances of the Book class for equality:

```
    Book firstBook  = new Book("0201914670");
    Book secondBook = new Book("0201914670");
    if (firstBook.equals(secondBook))
    {
        System.out.println("objects are equal");
    }
    else
    {
        System.out.println("objects are not equal");
    }
```

This program displays "objects are equal" even though `firstBook` and `secondBook` refer to two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the `equals` method if the identity operator is not appropriate for your class.

Note: If you override `equals`, you should override `hashCode` as well.

Examine the implementation of the `equals` method from the book's `Location` object. Note that the signature of the `equals` method must always be written the way that this code shows. The method returns a `boolean` and receives an `Object` as a parameter. In order to compare the fields of that object to the fields of the activating `Location` object, you have to cast the parameter as a `Location`. But first you have to be sure that the parameter actually *is* a `Location` object. After you have cast the parameter as a `Location`, Java will be able to get the values of that object's fields so you can compare them to the activating object's fields. You can refer to the fields without using a getter since this code is within the `Location` class.

```
public boolean equals(Object obj)
{
   if (obj instanceof Location)
   {
      Location candidate = (Location) obj;
      return (candidate.x == x) && (candidate.y == y);
   }
   else
      return false;
}
```

Assume that you have two `Location` objects named `loc1` and `loc2` (in code outside the `Location` class). Write a statement that prints "same" if those two locations have the same `x` and `y` values, and writes "different" otherwise. Use the `equals` method.

Think about a class that implements a container, such as the `IntArrayBag` class. The fields of `IntArrayBag` are an array of integers and a count of the number of integers that are currently stored in the array. What do you think would make two `IntArrayBag` objects equal?

hashCode

Hashing is an important topic in computer science, but we will not discuss it in this course. We will save that for CS 3460 where it will be given its due attention. Still, a brief discussion of the `hashCode` method in the `Object` class is in order here.

The value returned by the `hashCode` method in the `Object` class is the object's memory address in hexadecimal. By definition, if two objects are equal, their hash code must also be equal. If you override the `equals` method, you change the way two objects are equated and `Object's` implementation of `hashCode` is no longer valid. Therefore, if you override the `equals` method, you should also override the `hashCode` method. Since we are not going to do any hashing here in CS 2440, we will ignore this piece of information. You'll use it in CS 3460.

toString

The `toString` method is extremely useful, particularly in debugging. If you send an object reference to the `System.out.println` method, Java calls the `toString` method on that object and prints the `String` that gets returned. The `Object` class's `toString` method returns a `String` consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. This is not particularly useful. What would be better is to get a `String` containing the values of the fields of the object.

Here is the `toString` method in the Location class:

```
   public String toString( )
   {
       return "(x=" + x + " y=" + y + ")";
   }
```

What will the following code print?

```
    public static void main(String[] args)
    {
        Location loc1 = new Location(5, 7);
        Location loc2 = new Location(3, 9);

        System.out.println("Location 1 is " + loc1);
        System.out.println("Location 2 is " + loc2);
    }
```

Specifications, Designs and Implementations -- IntArrayBag (Section 3.2)

Section 3.2 of the text describes the specification, design, and implementation of a bag of integers. The most important thing about this section is to understand the meaning of the terms, "specification," "design," and "implementation." The `IntArrayBag` class gives us something with which to illustrate the concepts, and also serves as our first example of a container class.

What is a container class?

Here is an image of the first page of the **specification** of `IntArrayBag`.

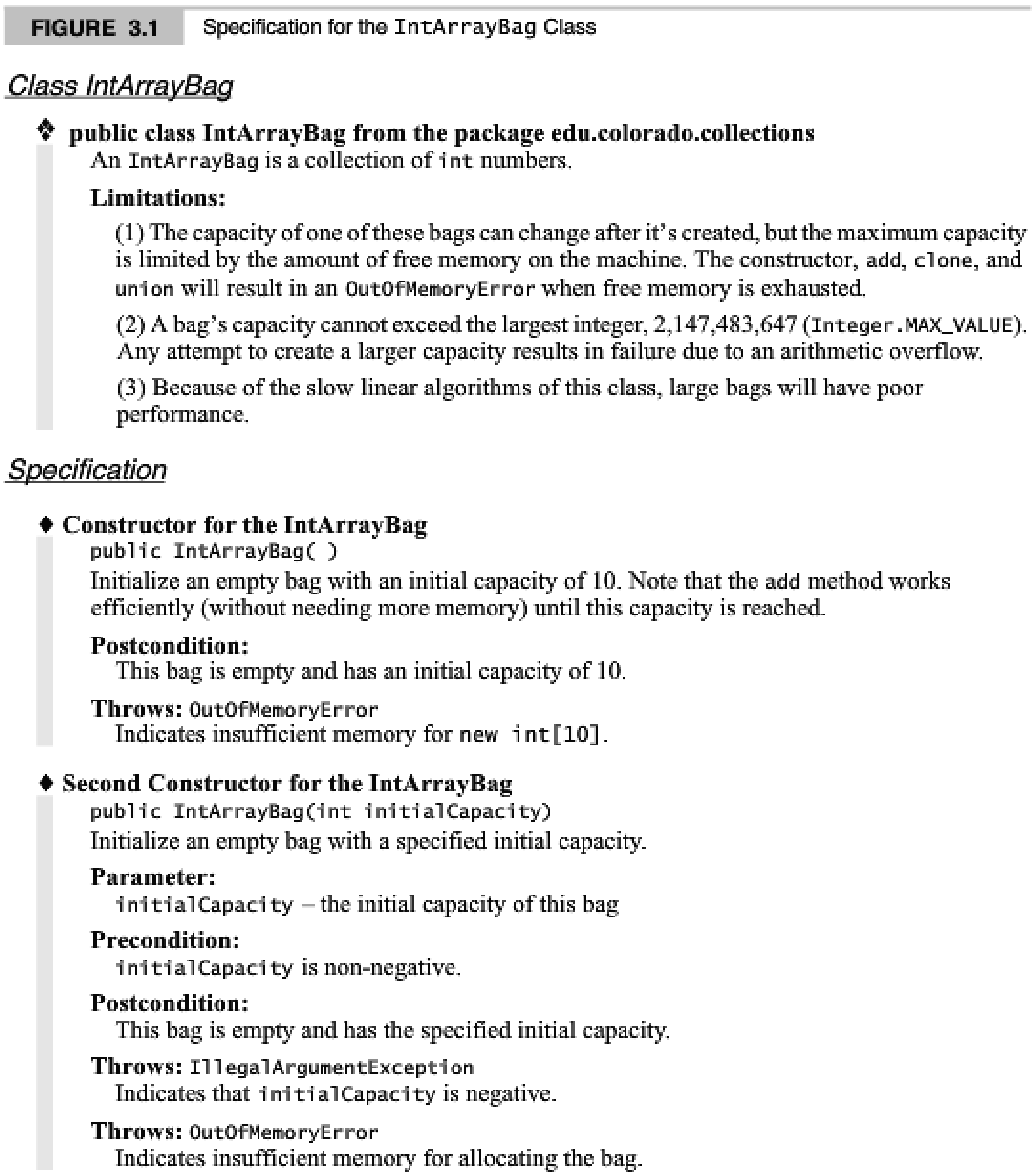


FIGURE 3.1 Specification for the `IntArrayBag` Class

Class IntArrayBag

❖ **public class IntArrayBag from the package edu.colorado.collections**

An `IntArrayBag` is a collection of `int` numbers.

Limitations:

(1) The capacity of one of these bags can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructor, `add`, `clone`, and `union` will result in an `OutOfMemoryError` when free memory is exhausted.

(2) A bag's capacity cannot exceed the largest integer, 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

(3) Because of the slow linear algorithms of this class, large bags will have poor performance.

Specification

♦ **Constructor for the IntArrayBag**

`public IntArrayBag( )`

Initialize an empty bag with an initial capacity of 10. Note that the `add` method works efficiently (without needing more memory) until this capacity is reached.

Postcondition:

This bag is empty and has an initial capacity of 10.

Throws: `OutOfMemoryError`

Indicates insufficient memory for `new int[10]`.

♦ **Second Constructor for the IntArrayBag**

`public IntArrayBag(int initialCapacity)`

Initialize an empty bag with a specified initial capacity.

Parameter:

`initialCapacity` – the initial capacity of this bag

Precondition:

`initialCapacity` is non-negative.

Postcondition:

This bag is empty and has the specified initial capacity.

Throws: `IllegalArgumentException`

Indicates that `initialCapacity` is negative.

Throws: `OutOfMemoryError`

Indicates insufficient memory for allocating the bag.

The specification continues over three additional pages in the text. There is a section for each method of the class similar to the sections shown for the constructors. What do those sections remind you of?

When you write a specification of a class, you first describe the things you want the class to do. Then you list the methods by which your class will do those things. For each method you write something like a full Javadoc comment. You include the method's signature, a description of all its parameters, the method's preconditions and postconditions, and a list of exceptions and errors that the method can throw.

The methods of the IntArrayBag class are

- `public IntArrayBag()`
- `public IntArrayBag(int initialCapacity)`
- `public void add(int element)`
- `public void addAll(IntArrayBag addend)`
- `public void addMany(int… elements)`
- `public IntArrayBag clone()`
- `public int countOccurrences(int target)`
- `public void ensureCapacity(int minimumCapacity)`
- `public int getCapacity()`
- `public boolean remove(int target)`
- `public int size()`
- `public void trimToSize()`
- `public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)`

What are these dots?

After you write a class specification, you can test your work by writing some code that will make use of the class once the class has been implemented. While you are writing this test program, you may make changes to your specification as you realize that it may not be quite what is needed to do the job.

At some point you will be satisfied with the specification. Note that you have not yet decided on the fields that your class will have. Nor have you decided on the algorithms that the methods will use to do the jobs you have specified for them. The first of these missing things, the fields, are added in the second phase.

The second phase of creating a new class is the design phase. At this point you decide what fields (instance variables) your class will need in order to do its work, and you write rules for how these fields are to be maintained. These rules are called the **invariant of the abstract data type**, and they form a contract that each method must live by. Each method (other than the constructors) can count on those rules being true when the method is called, and each method must guarantee that the rules are again true when the method finishes.

The instance variables of the `IntArrayBag` class are these:

- `private int[] data          // an array to store elements`
- `private int manyItems // how much of the array is used`

The invariant of the `IntArrayBag` class is as follows:

1. The number of elements in the bag is stored in the instance variable `manyItems`, which is no more than `data.length`.
2. For an empty bag, we do not care what is stored in any of `data`; for a non-empty bag, the elements of the bag are stored in `data[0]` through `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.

The third stage of the creation of a new class is the implementation phase. It is at this point that you figure out how you will write the code for each of your methods.

For each of these methods in the `IntArrayBag` class, briefly describe how our author decided to implement them.

- `public IntArrayBag()`
- `public IntArrayBag(int initialCapacity)`
- `public void add(int element)`
- `public void addAll(IntArrayBag addend)`
- `public void addMany(int… elements)`
- `public IntArrayBag clone()`
- `public int countOccurrences(int target)`

- `public void ensureCapacity(int minimumCapacity)`

- `public int getCapacity()`

- `public boolean remove(int target)`

- `public int size()`

- `public void trimToSize()`

- `public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)`

Why did the author decide to make the union method static?

Suppose we add the following `toString` method to the `IntArrayBag` class:

```
public String toString()
{
    String s = "";
    for (int i = 0; i < manyItems; i++)
    {
        s += data[i] + " ";
    }
    return s;
}
```

Keep track of what the `data` arrays and `manyItems` fields of `b1` and `b2` look like as the following code executes. Show the final picture and what is printed.

```
IntArrayBag b1 = new IntArrayBag();
IntArrayBag b2 = new IntArrayBag();
b1.add(1);
b1.add(2);
b1.add(3);
b2.addMany(4,5,6,7,8,9,10,11);
b1.addAll(b2);
System.out.println(b1.getCapacity());
System.out.println(b1);
System.out.println(b2);
System.out.println(IntArrayBag.union(b1, b2));
```

Do the same thing for bags b3 and b4 as the following code executes. Show the final picture along with what is printed.

```
IntArrayBag b3 = new IntArrayBag();
b3.addMany(25,25,10,5,25,10,5,1,1,25,25);
b3.add(b3.countOccurrences(25));
b3.remove(10);
b3.remove(10);
b3.remove(10);
IntArrayBag b4 = b3.clone();
b3.trimToSize();
System.out.println(b4.getCapacity() - b3.getCapacity());
System.out.println(b3);
System.out.println(b4);
```

DoubleArraySeq (Section 3.3)

We will need to look at section 3.3 of the text during this discussion of the `DoubleArraySeq` class. This is the class that you will be writing in the next lab, and it is very important that you understand exactly what the methods of the class are each supposed to do.

What are the differences between a Bag class and a Sequence class? In a Bag there is no significance to the ordering of the elements, and there is no way that someone using a Bag class can "step through" the elements of the Bag. With a Sequence, however, the order of the elements is at the user's control, and methods are provided to iterate through the elements from the beginning to the end.

The `DoubleArraySeq` class is a container class designed to hold doubles in a partially filled array. What is a partially filled array? How do you manage one?

Let's go through the specification of the DoubleArraySeq class. For each method, briefly note what the method is supposed to do.

`public DoubleArraySeq()`

`public DoubleArraySeq(int initialCapacity)`

`public void addAfter(double element)`

`public void addBefore(double element)`

```
public void addAll(DoubleArraySeq addend)
```

```
public void advance()
```

```
public DoubleArraySeq clone()
```

```
public static DoubleArraySeq concatenation(DoubleArraySeq s1,
DoubleArraySeq s2)
```

```
public void ensureCapacity(int minimumCapacity)
```

```
public int getCapacity()
```

```
public double getCurrent()
```

```
public boolean isCurrent()
```

`public void removeCurrent()`

`public int size()`

`public void start()`

`public void trimToSize()`

The design we will use makes use of three instance variables: (1) `data`, a partially filled array of doubles, (2) `manyItems`, a count of the number of doubles in the array, and (3) `currentIndex`, the index of the current element in the array (if there is one). The invariant of our abstract data type, `DoubleArraySeq`, is as follows:

1. The number of elements in the sequence is stored in the instance variable `manyItems`.
2. For an empty sequence, we do not care what is stored in any of `data`; for a nonempty sequence, the elements of the sequence are stored from `data[0]` to `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.
3. If there is a current element, then it lies in `data[currentIndex]`; if there is no current element, then `currentIndex` equals `manyItems`.

Under what circumstances is there no current element in a `DoubleArraySeq`?

After each of the following methods, what will the current element be, if any?

after a constructor	
after *addAfter*	
after *addBefore*	
after *addAll*	
after *advance*	
after *concatenation*	
after *removeCurrent*	
after *start*	

Under each of the following conditions, tell where `addBefore` and `addAfter` will place the items, and how the rest of the `data` array will change.

	addBefore()	*addAfter*()
sequence is empty		
there is a current item		
there is no current item		

Make a table for the time analysis of the Sequence operations similar to the one for the Bag operations on page 143.

Operation	**Time Analysis**	**Operation**	**Time Analysis**
Constructor (let c be initial capacity)		**start**	
addBefore without capacity increase and with current element		**advance**	
addBefore without capacity increase and without current element		**removeCurrent**	
addBefore with capacity increase		**isCurrent**	
addAfter without capacity increase and with current element		**getCurrent**	
addAfter without capacity increase and without current element		**concatenation(s1, s2)**	
addAfter with capacity increase		**getCapacity**	
s1.addAll(s2) without capacity increase		**ensureCapacity**	
s1.addAll(s2) with capacity increase		**size**	
clone		**trimToSize**	

In the lab you will be asked to add these methods to the author's specification:

1. `toString` method that returns a `String` representation of the `DoubleArraySeq`, and
2. `equals` method that determines if two `DoubleArraySeq` are equal.

The `toString` method must return a `String` in the following form. If the sequence is empty, the method should return "<>". If the sequence has one item, say 1.1, and that item is not the current item, the method should return "<1.1>." If the sequence has more than one item, they should be separated by commas, for example: "<1.1, 2.2, 3.3>." If there exists a current item, that item should be surrounded by square brackets. For example, if the second item is the current item, the method should return: "<1.1, [2.2], 3.3>."

The `equals` method should return true if the two sequences contain the same number of elements, if the corresponding entries in the sequence are the same, and if they have the same current element. The two sequences do not have to have the same capacity. The easiest way to determine whether or not two `DoubleArraySeqs` are the same is to compare their `String` representations. If the sequences are the same, their `Strings` will also be the same.

Trace what happens when the following code runs and show the final output.

```
public static void main(String[] args)
{
    DoubleArraySeq list = new DoubleArraySeq();
    list.addAfter(1.1);
    list.addBefore(2.2);
    list.addAfter(3.3);
    list.advance();
    list.advance();
    list.addAfter(4.4);
    list.start();
    list.addBefore(5.5);
    System.out.println(list.toString());
}
```

Inheritance (Sections 13.1, 5.1)

The material in this section is taken from our text book.

Extended classes use a concept called inheritance. In particular, once we have a class, we can then declare new classes that contain all of the methods and instance variables of the original class—plus any extras you want to throw in. This new class is called an extended class of the original class. The original class is called the superclass. The methods that the extended class receives from its superclass are called inherited methods.

There is another kind of access besides `public` and `private`. It is called `protected`. To outside classes that are not subclasses, `protected` is like `private`. But to extended classes, `protected` is like `public`.

Note that it is the `public` and `protected` members of the superclass that are accessible to the extended class. The private members are present – they have to be in order for the superclass to be able to do its job as it was designed – but they cannot be accessed directly by the extended class.

Extended classes may declare their own constructors or they may use a no-args constructor that is inherited from the superclass. Other superclass constructors with arguments are not inherited by extended classes. If an extended class has new instance variables that are not part of the superclass, then an inherited no-args constructor will set these new variables to their default values (zero for numbers, false for booleans, null for references) and then do the work of the superclass's no-args constructor. If an extended class declares any constructors of its own, then it does not inherit the no-args constructor of the superclass.

An object of the extended class may be used at any location where the superclass is expected. For example, suppose you have a superclass called `BankAccount` and several extended classes called `CheckingAccount`, `SavingsAccount`, `CDAccount`, etc. Java would allow you to declare a container of `BankAccount` objects and then add any instances of the subclasses to the container.

Sometimes an extended class needs to perform some method differently from the way that the superclass does. In this case, the extended class can contain a method with the same signature (or nearly the same – more on this in the next paragraph) but different code. This is called overriding an inherited method. Objects of the extended class type can still access the method in the superclass, but they have to qualify it with the name of the superclass and a dot prior to the method.

The data type of the return value of an overriding method doesn't have to be exactly the same type as the return value of the original method in the superclass. It can actually be a descendant of the data type of the return value of the original method. This is called a **covariant** return value.

Widening Conversions for Extended Classes

Assignments are allowed from an extended class to an object of the superclass type. Such assignments are called *widening conversions*. Widening conversions are always permitted.

When a program is running and a method is activated, the Java runtime system checks the data type of the actual object that activated the method, and uses the method from that type, rather than the method from the type of the reference variable. This technique of method activation is called **dynamic method activation** because the actual method to activate is not determined until the program is running. A method that behaves like this is called a **virtual method**. Java's nonstatic methods are always virtual.

Narrowing Conversions for Extended Classes

Assignments are also permitted from a superclass to one of its extended types. These are called *narrowing conversions*. When you write a narrowing conversion, you have to use a typecast or the compiler will report an error. The typecast will avoid syntax errors, but it is still possible to get an error at runtime if the object's type does not match the typecast.

Examples

1. Assume that `ComputerInputDevice` is a superclass of `MouseDevice`. The following statement occurs in a correct program:

   ```
   ComputerInputDevice device = new MouseDevice();
   ```

   What is the static type of `device`?

   What is the dynamic type of `device` after this statement?

2. The `Shape` class constructor has the following signature:

   ```
   public Shape(int xPos, int yPos)
   ```

   where `xPos` and `yPos` are the screen coordinates of the `Shape` and the constructor assigns them to fields of the `Shape` class.

   Write the class wrapper and constructor for a class called `Rectangle` which is a direct subclass of `Shape` and which contains two fields specifying a rectangle's width and length. The `Rectangle` constructor takes four parameters: the rectangle's x and y position coordinates, and its length and width. The `Rectangle` constructor's statements must cause a new `Rectangle` object to be initialized with x, y position and length and width.

What English phrase can be used to compare two things to see if there is an inheritance relationship between them?

What class is a superclass of all other classes?

In the following pairs of classes, identify the superclass by circling it, or write NONE if no inheritance relationship is clearly present.

- Horse, Animal
- Television, PowerCord
- Vehicle, Bus
- Page, Book
- Java, ProgrammingLanguage
- Mammal, Reptile
- Parrot, Bird
- Blue, Color
- Lawyer, Doctor

Assume we have four classes: `Person`, `Teacher`, `Student`, and `PhDStudent`. `Teacher` and `Student` are both subclasses of `Person`. `PhDStudent` is a subclass of `Student`. Which of the following assignment statements are syntactically legal, given the declarations shown?

```
Person p1;
Person p2;
PhDStudent phd1;
Teacher t1;
Student s1;
```

`p1 = new Student();`	legal	illegal
`p2 = new PhDStudent();`	legal	illegal
`phd1 = new Student();`	legal	illegal
`t1 = new Person();`	legal	illegal
`s1 = new PhDStudent();`	legal	illegal
`s1 = (Student) p1;`	legal	illegal
`s1 = (PhDStudent) p2;`	legal	illegal
`p1 = s1;`	legal	illegal
`t1 = s1;`	legal	illegal
`s1 = phd1;`	legal	illegal
`phd1 = s1;`	legal	illegal

Name one of the statements above that is an example of a widening conversion.

Name one of the statements that is an example of a narrowing conversion.

Abstract Classes and Interfaces (Sections 13.4, 5.5)

Abstract Classes

Sometimes, when you implement a large software system, you will have multiple classes that all have commonalities but that also have differences. In lab 6 you will implement a zoo of many different kinds of animals, for example. For the commonalities, you can make a superclass and put methods in it that all the subclasses will inherit. For the differences, each subclass will need to have its own implementation.

When you design the superclass, you know what methods you want each subclass to implement for themselves to handle the differences between them. You can make these methods *abstract* in the superclass which forces each subclass to implement them. For example, in the Animal class for lab 6, you will make abstract methods makeNoise and eat. To make a method abstract, you put the word abstract before the return type and you put a semicolon after the parameter list instead of implementing the method:

```
    /**
     * abstract method to say what happens when the animal makes noise.
     */
    public abstract void makeNoise();

    /**
     * abstract method to say what happens when the animal eats.
     */
    public abstract void eat();
```

Any class with abstract methods is an abstract class, so the class wrapper of the Animal class begins this way:

```
public abstract class Animal
```

If a class is abstract, you are not allowed to use the new keyword to create an object of that type. Its subclasses will not be abstract if they implement the missing methods (they will be *concrete*), and you will be able to create objects of the subclass types. On occasion, however, a subclass of an abstract class may need to be abstract itself. In the zoo lab, there is an abstract subclass of Animal called Canine, with four concrete subclasses of Canine called Dog, Wolf, Coyote, and WildDog. The abstract class, Canine, does not itself have any abstract methods. It is abstract only because the designer of this lab did not want users to be able to create new Canine objects.

Interfaces

An interface is something like a completely abstract class. It contains no concrete methods at all – it is simply a list of related methods that are specified but not implemented. A class can implement an interface by writing those methods. See the sample of an interface below.

There are many interfaces in the Java API such as ActionListener, Iterable, Iterator, and Comparable. You will create an interface for the zoo lab called Pet. Any class that implements the Pet interface (like Dog and Cat) will need to implement the play and beFriendly methods, as specified here:

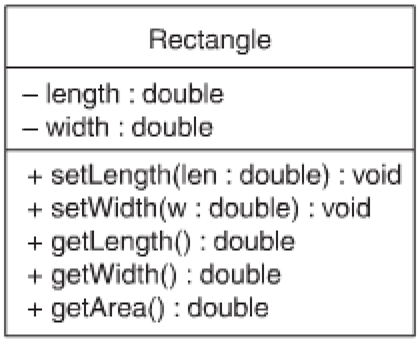
```
public interface Pet
{
    /**
     * method to say what pets do when they play
     */
    public void play();

    /**
     * method to say how pets are friendly.
     */
    public void beFriendly();
}
```

The Java keyword, instanceof, can be used on an object at runtime to see whether or not the object is of a type that implements the Pet interface. It is necessary to do that before you call either the play or beFriendly method on that object. Even after you confirm that an object implements the interface, you have to cast it before calling the methods, as shown here:

```
        for (Animal a : zooAnimals)
        {
            System.out.println(a.getName());
            a.sleep();
            System.out.println(a.getName() + " is hungry!");
            a.makeNoise();
            a.eat();
            a.roam();
            if (a instanceof Pet)
            {
                ((Pet) a).play();
                ((Pet) a).beFriendly();
            }
            System.out.println();
        }
```

UML (Unified Modeling Language) Diagrams

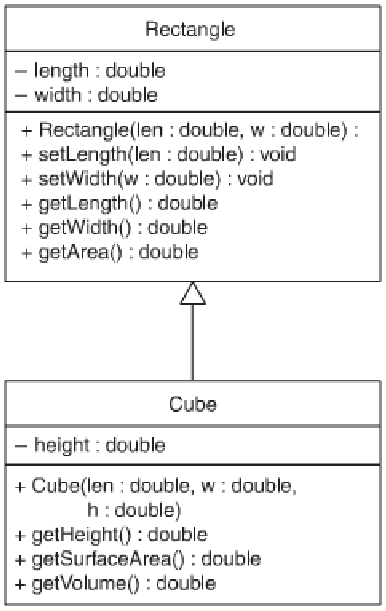
UML diagrams are standard diagrams for describing object-oriented systems. A sample UML diagram for a class called `Rectangle` is shown here:



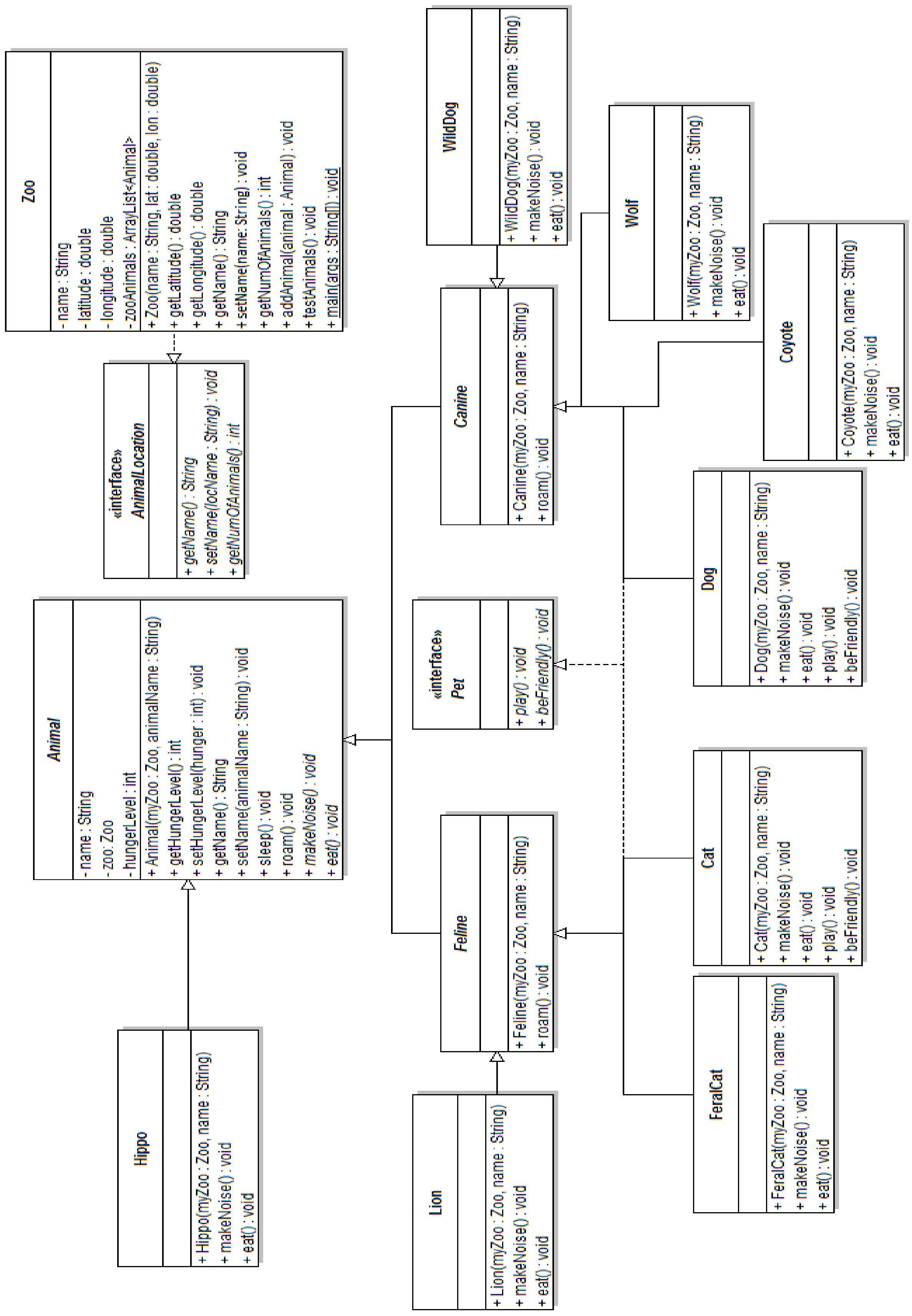
Notice that the diagram is a box that is divided into three sections. In the top section you write the name of the class. In the middle section you list the class's fields, and in the bottom section you list the class's methods. Although it is optional, in this example the names of the fields and methods are preceded by either – or +. The – indicates that the member is private while + indicates that it is public.

After the name of a field you place a colon and then write the field's type. After the name of a method you put a parameter list, then a colon, and then the return type of the method. Within the parameter list you put the name of each parameter followed by a colon and its type. Multiple parameters are separated with commas.

To illustrate inheritance in UML diagrams, a subclass is connected to its superclass with a line that has an open arrowhead at one end, like this:


Rectangle
– length : double
– width : double
+ Rectangle(len : double, w : double) :
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double
Cube
– height : double
+ Cube(len : double, w : double, h : double)
+ getHeight() : double
+ getSurfaceArea() : double
+ getVolume() : double

In order to indicate in a UML diagram that a class is abstract, put the name of the class in italics and write the specification of the abstract methods in italics. In order to indicate an interface, put the notation "`<<interface>>`" in the title area of the UML box. Here is the UML diagram of the classes from lab 6.



Study List for 2440 Exam 1

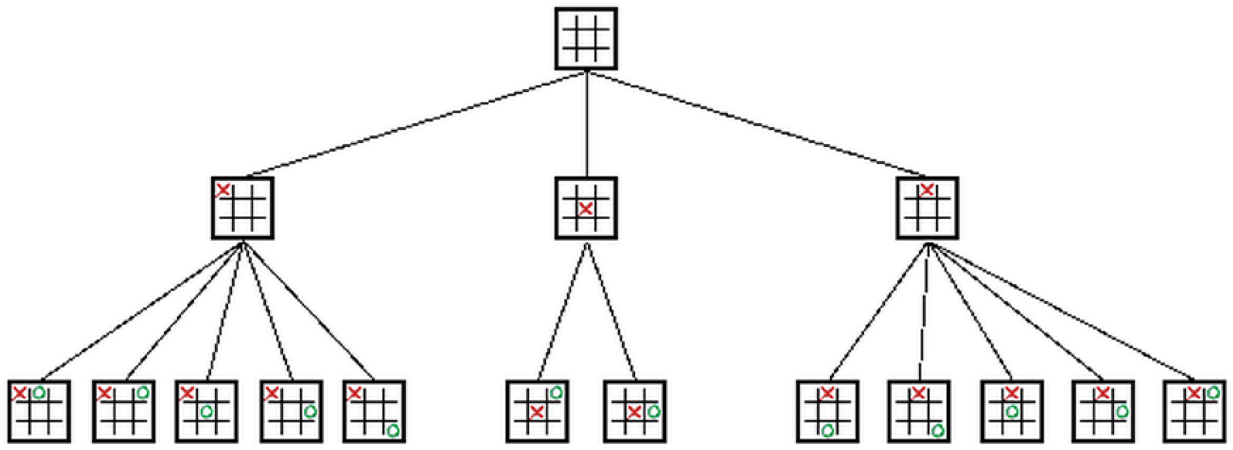
1. From 1440: Be able to trace code that calls methods, passes values, returns values, evaluates expressions, and traverses arrays. Understand the difference between declaring a variable and allocating an object using "new."
2. Understand the behavior of objects and primitives when passed as arguments to methods.
3. Know how to read and write methods that process two-dimensional arrays.
4. Know how to create and use enumerated data types.
5. Know the difference between specification, design, and implementation.
6. Know how to write preconditions and postconditions for methods. What is the value of preconditions and postconditions? At what point in the creation of a new class are they written?
7. Know the formula used to predict running time for an algorithm on a second data size, given the big-O analysis of the algorithm and given one data size/running time combination.
8. Be able to determine the big-O of methods (like you did in the second homework).
9. Understand the effects of making a method or a field static.
10. Know how to write an `equals` method, a `clone` method (for classes that need deep copies and those that do not), and a `toString` method.
11. Be able to identify the parts of a GUI and understand the Java classes that are used to create those parts.
12. Know how exception handling works. Be able to both catch and throw exceptions, and know when it is appropriate to do both.
13. Know how to shift data in an array to either make room for a new element or remove an element.
14. Understand the concept of a partially-filled array and how one is maintained.
15. Know what an invariant is. Who writes the invariant and at what stage?
16. Understand the invariant of the `IntArrayBag` class, and know how the `IntArrayBag` class methods maintain that invariant.
17. Understand the invariant of the `DoubleArraySeq` class, and know how the `DoubleArraySeq` class methods maintain that invariant. Be able to trace code making use of `DoubleArraySeq` methods.
18. Be able to explain the big-O analyses of the methods in the `IntArrayBag` and `DoubleArraySeq` classes.
19. Given a UML diagram for a set of related classes and interfaces, such as the UML diagram for the `Zoo` lab, be able to write the classes.

AbstractGame Class (Section 13.4)

In chapter 13, the author provides an abstract class with which we can implement any 2-player game with perfect information in which a human player competes against the computer. A game with perfect information is one in which both players know the current status of the game. Chess and checkers are such games – both players can see the game board and players have no secrets from each other. Card games in which players cannot see each other's hands, however, do not have perfect information.

`AbstractGame` implements an AI algorithm called minimax that searches the nodes of a game tree for the best move to make whenever it is the computer's turn to play. It involves looking several moves ahead and predicting that the human player will be smart and make the best moves he or she can make (the player will minimize the computer's chances of winning). Under that assumption, the computer makes the best move it can make to maximize its chances of winning. The more moves ahead you set the algorithm to look, the harder the computer is to beat.

Here is a game tree that shows the first two moves in a game of Tic-Tac-Toe. The tree has been simplified by removing symmetrical positions.



We will look at some online descriptions of the algorithm in class. The algorithm is a bit complicated to implement. What's nice is that we don't have to implement it – the author already did that.

In order to write a 2-player game (human vs. computer), we will extend `AbstractGame`. But we also need to design our own implementation of whatever game we want to create. We first have to decide what data structures we need to represent the state of the game at any point in time. These data structures will be the fields of our game class. We will need a constructor to initialize these data structures at the start of a game. We will need a clone method that will correctly make a deep copy of our game. We also have to write six methods that are abstract in `AbstractGame`:

- *`computeMoves(): Vector<String>`*
  This method must generate a Vector of legal moves that can be made at any given point in the game. Moves must be represented as Strings. AbstractGame will call on this method whenever it needs to know what moves are possible for the current player in its search of the game tree.
- *`displayStatus(): void`*
  This method prints the game to the screen so that the player can see the current state of the game.
- *`evaluate(): double`*
  This method needs to return a numeric value that correlates to the quality of the current game state from the point of view of the computer. The method needs to return a positive number when the game state favors the computer – the better the game state is for the computer, the higher the number that this method returns. The method needs to return a negative number if the game state is better for the human, and the better the game state is for the human, the more negative the return value should be.
- *`isGameOver(): boolean`*
  This method returns true if the game is over and false otherwise.
- *`isLegal(move: String): boolean`*
  This method is passed a move. It returns true if that move is legal, given the current state of the game and whose turn it is.
- *`makeMove(move: String): void`*
  This method is responsible for changing the game state when either the computer or the human makes a move. The move to be made is passed to the method.

The author provides us with a sample game, `Connect4`, that we can use to help understand how to make use of `AbstractGame`. We will examine the `Connect4` game during class. For the next lab you will implement a `Wari` game. We will spend more time discussing `Wari` right before the lab.

Linked Lists (Sections 4.1, 4.2)

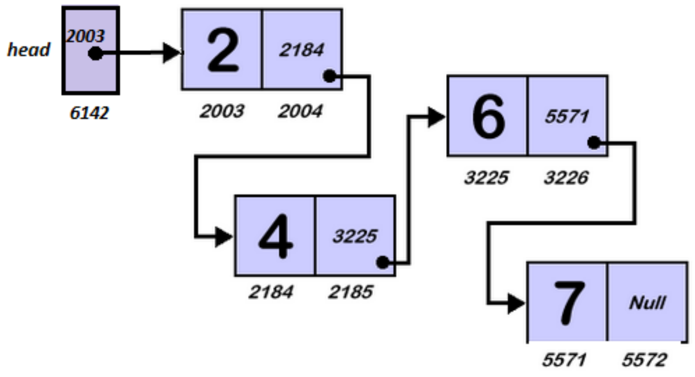
The concepts involved in using linked lists to create data structures are perhaps the most important concepts of a Computer Science II class. It is imperative that you get a good understanding of this material because it is foundational to many different data structures that you will use in 3460 and afterward.

The container classes we have seen so far, `IntArrayBag` and `DoubleArraySeq`, were implemented with arrays. Container classes can also be built with linked lists. We need to understand the differences between arrays and linked lists at a conceptual level before we look at the implementation details of linked lists.

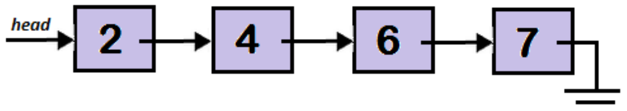
Arrays are stored in consecutive cells in RAM. When you create an array you have to specify a type and a count of the number of elements. Java knows how much space one element of that type takes, so it multiples that size times the number of elements you want, and allocates a block of memory of that total size for the array. Random access is possible because array elements are stored consecutively. A program can go directly to any element in an array by using the element's index. Java knows where the array begins in RAM. When you want to access element i of a single-dimension array, Java multiplies i by the size of one element and adds that amount to the address of element 0.

Random access is the primary benefit of arrays. One drawback is that the size of an array is fixed. Another drawback is that it is expensive to insert a new element into an array or to delete an element – all elements from that point forward have to be shifted.

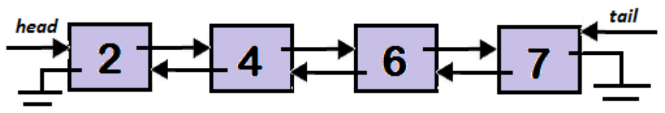
Linked lists are composed of structures that we call nodes. In a singly-linked list, each node contains both data and the address in RAM of the next node in the structure. These addresses are what we call *links*. The last node in the list has a null link to mark it as the end of the list.



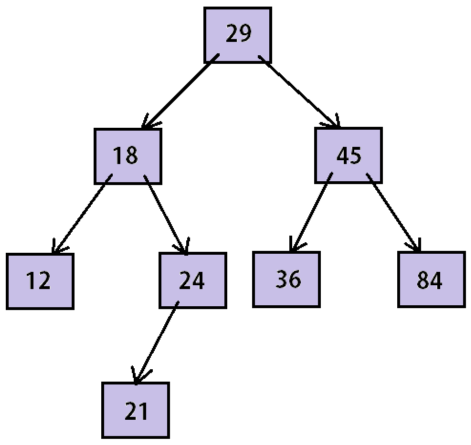
Typically we simplify the picture of a linked list by removing the addresses and leaving arrows to indicate the links. We use a "ground" symbol to indicate null.



Linked lists can be created in such a way that each node contains a link both to the next node in the sequence and to the previous node. We call these doubly-linked lists.



Binary trees of nodes can be created by putting links to left and right children in each node along with data. The following picture is simplified even further by leaving out all the null links in nodes that have missing children.



We will not implement doubly-linked lists or trees in CS 2440, but you will implement them in later courses. For now we will work only with singly-linked lists.

Recall that when you begin the creation of a new class, the first phase is specifying all the methods that your class will need. The list of methods from the `DoubleArraySeq` class included `addBefore`, `addAfter`, `removeCurrent`, `start`, `advance`, `isCurrent`, `getCurrent`, and others. At the second phase you decide what data structures you will use. For `DoubleArraySeq` we used an array of `doubles` named `data`, a count named `manyItems`, and an index of the current element named `currentIndex`.

Soon you will implement another class called `DoubleLinkedSeq`. It will have almost the exact same specification as `DoubleArraySeq`, but at the design phase you will choose to use a linked list to contain the sequence of `doubles` instead of an array. The data members of `DoubleLinkedSeq` are references to nodes, along with a count of the number of nodes in the list.

Note that to someone using either of your sequences as a container for their data, they could write a program for one of your classes and it would work perfectly well on the other one (other than the declaration of the sequence). All of the methods do exactly the same things in either implementation.

The StringNode Class

We will first examine a `StringNode`, a node that stores a `String` for its data. Our node has two fields. One is for the data portion and the other is a link (reference) to the `StringNode` that is next in the linked list. The class has various constructors, getters, and setters as shown here:

```
public class StringNode
{
    private String data;
    private StringNode link;

    public StringNode()
    {
        data = null;
        link = null;
    }

    public StringNode(String data)
    {
        this.data = data;
        link = null;
    }

    public StringNode(String data, StringNode link)
    {
        this.data = data;
        this.link = link;
    }

    public void setLink (StringNode newLink)
    {
        link = newLink;
    }

    public void setData (String newData)
    {
        data = newData;
    }

    public StringNode getLink ()
    {
        return link;
    }

    public String getData ()
    {
        return data;
    }
}
```

Now consider the following code:

```
public static void main(String[] args)
{
    StringNode str = new StringNode();
    str.setData("Dino");
    StringNode  str2 = new StringNode("Fred", str);
    StringNode str3 = new StringNode("Barney", str2);
    StringNode str4 = str3;
    while (str4 != null)
    {
        System.out.println(str4.getData());
        str4 = str4.getLink();
    }
}
```

To figure out what the code will output, we need to draw a picture.

What is `str`?

What is `str.getData()`?

What is `str.getLink()`?

What is `str2`?

What is `str2.getData()`?

What is `str2.getLink()`?

What is `str2.getLink().getData()`?

What does the code output?

Now write some code to make a new `StringNode` with "Fido", and insert it between "Fred" and "Dino".

To check for the equality of references, you may use `==`. The equality operator, `==`, tests whether two reference variables contain the same address. To check for the equality of data in the nodes that are being referenced, you will need to use the `equals` method unless the data is a primitive, such as `int` or `double`.

We have seen a class that defines one node. A linked list class has to manage a series of nodes. The simplest form of a linked list will have a field that is a count of the number of nodes in the list, and a reference to the first node in the list. To get to the other items in the list we have to *traverse* the list using the link references the way we did above. Suppose the fields of a list are called `head` and `manyNodes`. Here are common ways to traverse, printing the list:

`while` loop:

```
      Node traverse = head;
      while (traverse != null)
      {
          System.out.println (traverse.getData());   // print the actual
      data portion
          traverse = traverse.getLink();
      }
```

`for` loop:

```
      for (Node traverse = head; traverse != null; traverse =
traverse.getLink())
      {
          System.out.println (traverse.getData());
      }
```

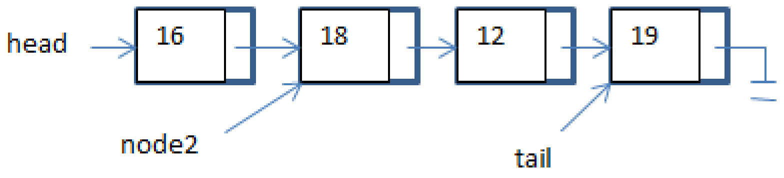
Let's practice tracing some code using the `StringNode` class. Draw a picture showing the results of the following code and write what the code outputs.

```
public class Weather
{
    public static void main(String args[])
    {

        String arr[] = {"snow", "rain", "sleet"};
        StringNode p, q, r, s;

        p = new StringNode("hail");
        for (int i = 0; i < 3; i++)
        {
            p.setLink (new StringNode(arr[i], p.getLink()));
        }
        System.out.println("Answer 1 ");
        for (r = p; r != null; r = r.getLink())
        {
            System.out.println(r.getData());
        }

        r = p;
        while (r.getLink() != null)
        {
            r = r.getLink();
        }

        r.setLink(p);
        q = r.getLink();
        q = q.getLink();
        System.out.println("Answer 2 " + q.getData());

        s = new StringNode("cloudy", q);
        r = r.getLink();
        r.setData("snow");
        r.setLink(s);
        System.out.println("Answer 3 " + q.getData());
        System.out.println("Answer 4 " + p.getData());

        System.out.println("Answer 5 ");
        while (s != p)
        {
            System.out.println(s.getData());
            s = s.getLink();
        }
    }
}
```

The IntNode Class from Our Textbook (Section 4.3)

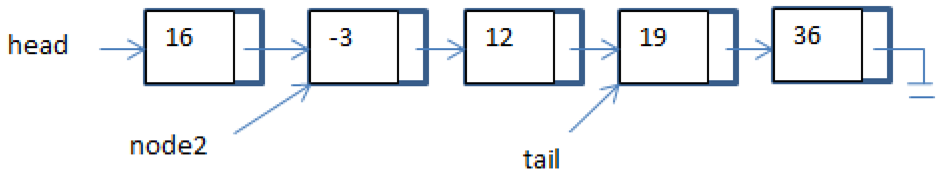
The author of the textbook provides an `IntNode` class that contains more methods than our simple `StringNode` class contains. Along with a constructor and get and set methods for the two fields in the `IntNode`, the author adds methods that can be used to manage a linked list. You will make use of this `IntNode` class when you write the `DoubleLinkedSeq` class.

Suppose we have created the following list of `IntNodes`:



Then the code shown here will change the list as you see below:

```
node2.setData(-3);
tail.addNodeAfter(36);
```



Here is the code of the `addNodeAfter` method. We activated the method on the node that `tail` refers to. The method made a new `IntNode`, put 36 in the `data` field, set the `link` field to the same value as the `link` of the `tail` node, and then changed the `link` of the `tail` node so that it referred to the new `IntNode`.

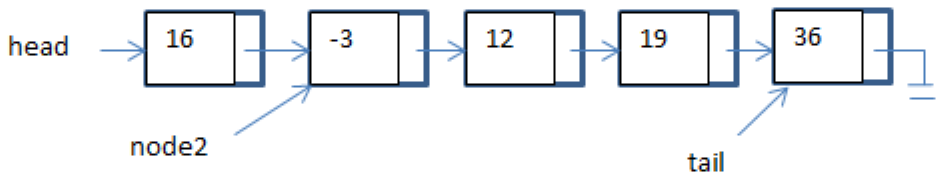
```
public void addNodeAfter(int item)
{
   link = new IntNode(item, link);
}
```

Since we want to maintain the tail pointer on the last node of the list, we need to move tail to the node that was just added. Write some code to do that:

Another method that the author provides in the `IntNode` class is called `removeNodeAfter`. This method can remove the node that comes after the activating node. Here is the code of the `removeNodeAfter` method:

```
public void removeNodeAfter( )
{
   link = link.link;
}
```

Here is the current list:



Draw a picture of the list if we call the `removeNodeAfter` method with `head`:

What happens to the node that `node2` refers to?

The author also provides some static methods that can be used by classes that create linked list structures. One of these methods is called listSearch. You pass it a reference to the node where you want the search to begin (usually the head node) and a target to search for. The method returns a reference to the node containing the target or null if the target is not found. Here is the method:

```
public static IntNode listSearch(IntNode head, int target)
{
   IntNode cursor;

   for (cursor = head; cursor != null; cursor = cursor.link)
      if (target == cursor.data)
         return cursor;

   return null;
}
```

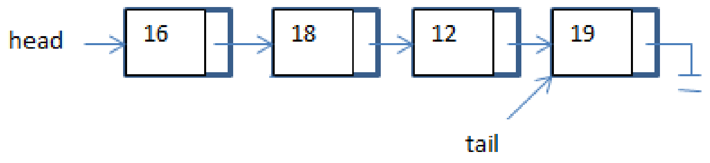
Notice that in static methods, IntNode objects are passed as parameters. There is no activating object. To call a static method we have to use the name of the class. How would we call the listSearch method to search for 19 in the above list?

Other static methods in the IntNode class are:

- `public static IntNode listLength(IntNode head):` Returns the number of nodes in the list.
- `public static IntNode listPosition(IntNode head, int position):` Returns a reference to the node whose "index" is equal to the parameter, position. Unfortunately, the author decided to make the head node be position "1".
- `public static IntNode listCopy(IntNode source):` Copies the list and returns the head pointer to the copy.
- `public static IntNode[] listCopyWithTail(IntNode source):` Copies the list and returns both the head pointer and the tail pointer to the copy (in an array).
- `public static IntNode[] listPart(IntNode start, IntNode end):` Copies the part of the list from start to end and returns a head pointer and tail pointer to the copy (in an array).

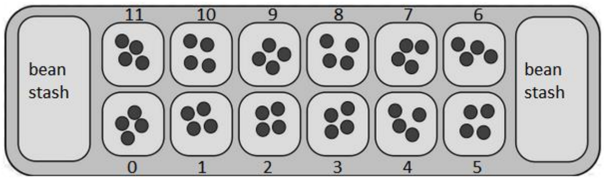
Here is one of the copy methods. Trace its behavior on the small list below the code.

```
public static IntNode[ ] listCopyWithTail(IntNode source)
{
    IntNode copyHead;
    IntNode copyTail;
    IntNode[ ] answer = new IntNode[2];

    // Handle the special case of the empty list.
    if (source == null)
    {
        return answer; // The answer has two null references .
    }

    // Make the first node for the newly created list.
    copyHead = new IntNode(source.data, null);
    copyTail = copyHead;

    // Make the rest of the nodes for the newly created list.
    while (source.link != null)
    {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
    }

    // Return the head and tail references.
    answer[0] = copyHead;
    answer[1] = copyTail;
    return answer;
}
```



Wari Game

Wari is a game for two players, using a 2 x 6 board and 48 beans.



Rules of the game:

1. Set-up: At the beginning of the game, four beans are placed in each of the 12 squares on the board. The board is placed between the two players. The six squares on Player A's side of the board are Player A's squares, and similarly for Player B. Players decide who will go first.

2. Players alternate turns. On a turn, a player selects one of his or her squares, removes all the beans from that square, and redistributes them counter-clockwise around the board, one bean per square. If there are enough beans to go all the way around the board and back, the starting square is skipped. The starting square is always empty after redistribution.

3. If during redistribution the last bean is placed in an opponent's square so that the number of beans in this square becomes 2 or 3, then the player captures the beans from that square, and also captures any beans from consecutive preceding opponent's square(s) where the number of beans in those squares is 2 or 3. The captured beans are placed in the player's bean stash at the end of the board.

4. There are two ways to win the game. The game stops when one player captures 25 or more beans and that player is the winner. The game also stops when, at a player's turn, there are no beans in any of this player's squares. At that point the player with the most beans is the winner.

Preparation for Lab:

In lab, you will be implementing `Wari` by extending the `AbstractGame` class provided by the author of our textbook. Recall that in order to extend the AbstractGame class, the subclass must provide the following:

1. Instance variables (fields) to represent the status of the game being written.
2. A way of representing a move in the game as a String.
3. A constructor that initializes the instance variables for the start of a game after it calls the default no-arg superclass constructor.
4. These methods that are abstract in `AbstractGame`:
    a. `protected Wari clone()`: You need to do a deep copy in your clone method.

b. `protected Vector<String> computeMoves():` Find out whose turn it is and put all the String representations of valid next moves for that player into a Vector of Strings. Return the Vector.
c. `protected void displayStatus():` Display the current status of the game.
d. `protected boolean isGameOver():` Return whether or not the game is over.
e. `protected boolean isLegal(String move)`: Return whether the parameter, move, is a legal move to be played next.
f. `protected void makeMove(String move)`: Make this move, updating your fields as necessary.
g. `protected double evaluate():` Return a value estimating how good the current status of the game looks for the computer – the higher the value, the better for the computer. Zero means it looks even at this point. A negative value means the computer appears to be losing.

5. A main method that calls `repeatPlay` and passes "Wari" and `DEPTH`. Make `DEPTH` a `final` variable in your game class.

What fields might you use for the `Wari` game? As you consider the possibilities for representing the board, think about how easy or hard it will be to redistribute beans using each of the representations.

How will you represent moves as `Strings`?

How will you determine which moves are legal at any point in time?

Sketch out what your `displayStatus` method will print.

How will you write the `makeMove` method? This is the method that has to redistribute beans and then collect any beans that the player captures.

Be familiar with the methods that `Wari` inherits from `AbstractGame`. They are listed in the book, so be sure to bring your book to lab. One very useful method is `nextMover`. It returns whose turn it is, either `Player.human` or `Player.computer`.

The IntLinkedBag Class (Section 4.4)

The specification for the `IntLinkedBag` class is the same as for the `IntArrayBag class`. We have a constructor and these methods in the class:

- `public void add(int element)`
- `public void addAll(IntLinkedBag addend)`
- `public void addMany(int... elements)`
- `public IntLinkedBag clone()`
- `public int countOccurrences(int target)`
- `public int grab()`
- `public boolean remove(int target)`
- `public int size()`
- `public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)`

The IntLinkedBag class has two instance variables, `head` and `manyNodes`, with this invariant:

1. The elements in the bag are stored in a linked list.
2. The head reference of the list is stored in the instance variable `head`.
3. The total number of elements in the list is stored in the instance variable `manyNodes`.

What values should `head` and `manyNodes` have when the bag is empty?

Why do we have a field for `manyNodes` when there is a method in the `IntNode` class to compute the length of a list? There is a `size` method in the `IntLinkedBag` class. Which should it use?

Does the `IntLinkedBag` class need to do a deep copy in the clone method, or will simply calling `super.clone` suffice?

The `IntLinkedBag` class uses methods from the IntNode class for many of its tasks. What method from the `IntNode` class would help with the `clone` method? How should the method be called?

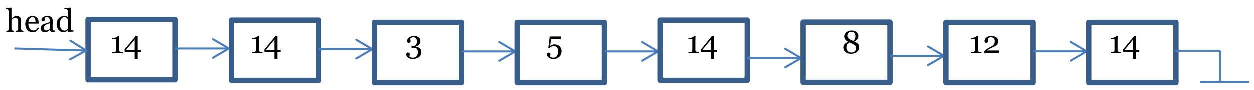
Recall that the parameter to a `Bag's remove` is the item to remove, and that item must be found in the Bag before it can be removed. What method from the `IntNode` class would help with finding the element to be removed? How is it called?

Remember that to remove an element from an `IntArrayBag`, we overwrote it with the last element in the array and reduced `manyNodes` by one. Once the element to be removed is found in the `IntLinkedBag`, what is the author's algorithm for removing the item? State this in English first, and then list the steps in Java.

Another way the remove could have been done was to change the link of the previous node to the link to the following node. What would have been difficult about this?

The `countOccurrences` method has an `int` parameter, and the method is supposed to return the number of occurrences of that `int` in the list. Your textbook uses the `IntNode's listSearch` method to do this. Trace this code on the following list when the method is told to count 14's.

```
public int countOccurrences(int target)
{
   int answer;
   IntNode cursor;

   answer = 0;
   cursor = IntNode.listSearch(head, target);
   while (cursor != null)
   {  // Each time that cursor is not null, we have another occurrence of
      // target, so we add one to answer and then move cursor to the next
      // occurrence of the target.
      answer++;
      cursor = cursor.getLink( );
      cursor = IntNode.listSearch(cursor, target);
   }
   return answer;
}
```



Write the method yourself a different way without using the `listSearch` (just traverse the list and compare each piece of data and update the count accordingly).

What are the possible values of the expression: `(int)(Math.random() * manyNodes ) + 1`?

Using `Math.random`, write an expression that calculates a random integer between -50 and 40.

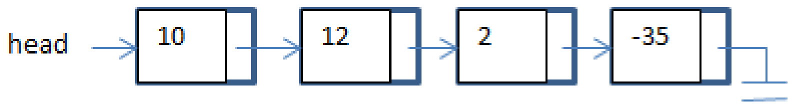
What does the `grab` method do?

Where are new elements added to the `IntLinkedBag`? Why?

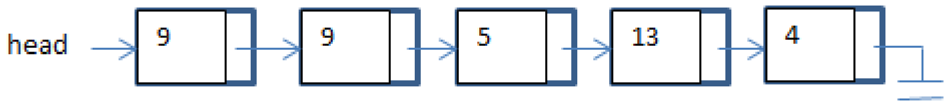
Both the `union` and `addAll` methods put two bags together. How is the `union` method different from the `addAll` method?

Suppose we have a program that has two `IntLinkedBag` objects `bagA` and `bagB` as shown:

`bagA`:
`manyNodes` = 4



`bagB`:
`manyNodes` = 5



What would `bagA` look like after the following two lines of code were executed?

```
bagA.addAll(bagB);
bagB.remove(5);
```

For each of the following `Bag` methods, tell the BigOh of the linked implementation. Is each method more or less efficient than the array implementation?

	`IntLinkedBag`	compared to `IntArrayBag`
`add(int item)`		
`remove (int item)`		
`clone()`		
`size()`		
`grab()`		
`addMany(int … elements)`		
`countOccurrences(int item)`		

Name one advantage of the linked bag over the array bag?

Name one advantage of the array bag over the linked bag?

The DoubleLinkedSeq Class (Section 4.5)

In lab, you will be implementing the double sequence as a linked list. Your textbook suggests that you use five fields: `head`, `tail`, `cursor`, `precursor`, and `manyNodes`.

What does each of the five fields represent?

- `head`:
- `tail`:
- `cursor`:
- `precursor`:
- `manyNodes`:

Look back at the invariant for `IntLinkedBag` and use it as a model to write an invariant for the `DoubleLinkedSeq` class:

Suppose you have the following sequences. Draw the linked structure and show the values of the fields: `head`, `tail`, `cursor`, and `precursor`:

<3.2, 4.1, 5.5, [1,0], 3.1>

<>

<4.3>

<[4.3]>

<[5.5], 6.2>

`isCurrent:` One of the methods you will write is named `isCurrent`. It returns true if there is a current item, and false otherwise. How can you detect if there is no current item?

`addBefore`: Here is a case by case analysis for the `addBefore` method. For each case, state which of the four reference fields will have to change. Assume that we check the conditions in the order they are listed.

if the list is empty

---

else if there is no current item or the first item is the current item

---

else (there are at least two items in the list, there is a current item, and it is not the first item)

Write code for the part of `addBefore` that deals with the case that the list is empty.

Write code for the part of `addBefore` that deals with the case that either there is no current item or the current item is the first item.

Write code for the last case, when the current item is not the first one.

`removeCurrent:` Great care must be taken to check all cases when removing the current element from the sequence. Draw pictures and take note of which fields change under the following circumstances.

There is only one element in the sequence?

The current element is first?

The current element is at the end?

All other cases? (There are more than two nodes, and the current element is somewhere between the first and last.)

The `DoubleLinkedSeq` class adds a tail reference that was not part of the `IntLinkedBag` class. What methods of the sequence class make it useful to keep a tail reference?

addAfter: What cases must be considered in the implementation of the `addAfter` method? List them in the order you will check them and for each case state which references will have to change?

You will have a `DoubleNode` class that has helper methods similar to the `IntNode` class. What helper method from the `DoubleNode` class will help with the concatenation method?

Consider the following methods in the `DoubleLinkedSeq` class. What is the complexity of the method? Is the method more or less efficient than the array implementation?

	DoubleLinkedSeq	DoubleArraySeq
addBefore(double element)		
addAfter(double element)		
removeCurrent ()		
start()		
isCurrent()		
getCurrent()		
advance()		
clone()		
size()		
concatenation(DoubleLinkedSeq seq1, DoubleLinkedSeq seq2)		
addAll(DoubleLinkedSeq other)		
toString()		
equals(DoubleLinkedSeq seq)		

The `clone` method requires special consideration. See the discussion on page 235 to help with this method. For each of the following cases, describe how you will use the `DoubleNode` copy methods to copy the sequence and how you will set the reference fields.

if the sequence to be copied has no current element:

else if the current item is the first item

else (the current item comes after the first one)

Generics (Sections 5.1-5.3)

There are eight primitive types in Java. Name them:

1. ______________ 2. ______________ 3. ______________ 4. ______________

5. ______________ 6. ______________ 7. ______________ 8. ______________

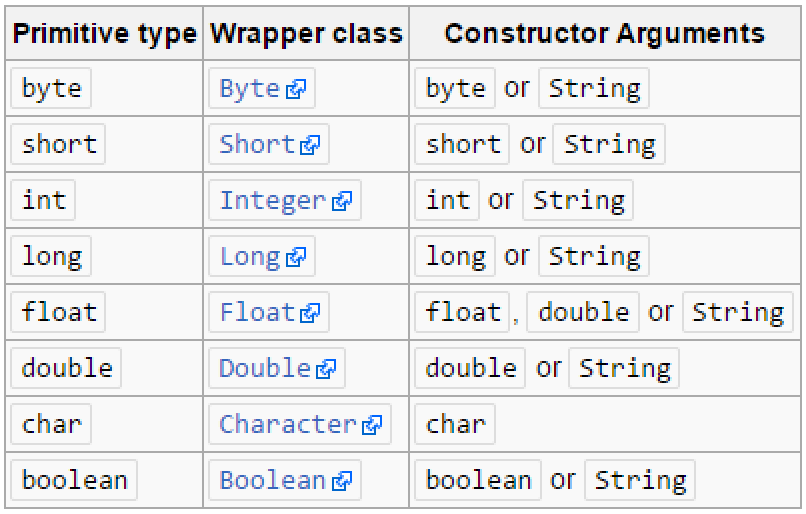
Everything else is an object. All objects are part of the hierarchy of Java objects and are all subclasses of the `Object` class. We have already talked about the methods of `Object` that are inherited by all other classes but that generally have to be overridden to be useful. Name four such methods:

1. ______________ 2. ______________ 3. ______________ 4. ______________

In the Java library there are many useful container classes such as `ArrayList`, `Vector`, and `LinkedList`. These are all generic classes. When you declare one, you have to tell Java what type of objects you intend to store in the container (and only objects will work). For example, here is the declaration of a `Vector` of `Strings`:

```
        Vector<String> moves = new Vector<String>();
```

The type is supplied in angle brackets. Any type other than the eight primitives will work. But what if you want to store ints or doubles or one of the other primitives in a generic library class like `ArrayList` or `Vector`? In that case, you have to use the wrapper class for the type of primitive you need to store. Each one of the Java primitive types has a



Primitive type	Wrapper class	Constructor Arguments
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double` or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

corresponding wrapper type. From Wikipedia:

An object of type `Byte` stores one byte, and similarly for each of the other wrapper classes. Conveniently, Java allows you to treat a variable of a wrapper type as if it was a primitive when you refer to it in your code by autoboxing and auto-unboxing your wrapper objects.

For example, in the following code, `x` is assigned the value 5 using the keyword `new` to create a new `Integer` object and passing 5 to the constructor. `Integer` y, on the other hand, is assigned the value 7 just as if `y` was an `int` instead of an `Integer`. This is an example of autoboxing. The next line causes both autoboxing and auto-unboxing to occur. In order to add x and y, Java has to get the `ints` 5 and 7 out of the `Integer` objects. It does this whenever an `Integer` object is used in an arithmetic expression, and it's called auto-unboxing. Once the `ints` are added together, Java needs to box up the sum in order to put it into the `Integer`, `z`. This is another example of autoboxing.

```
public class Wrappers
{
    public static void main(String[] args)
    {
        Integer x = new Integer(5);
        Integer y = 7;
        Integer z = x + y;
        System.out.println(x + " " + y + " " + z);
    }
}
```

A disadvantage of using wrapper classes is that there is some overhead involved in their usage, and that extra work for Java causes the code to run a little slower. An advantage is that you can turn primitives into objects when you want to put them in generic containers.

We will first look at generic methods. Then we will examine how you write your own generic classes.

Generic Methods

Suppose you want to reverse the elements of an integer array. Here is some pseudocode that will do the job:

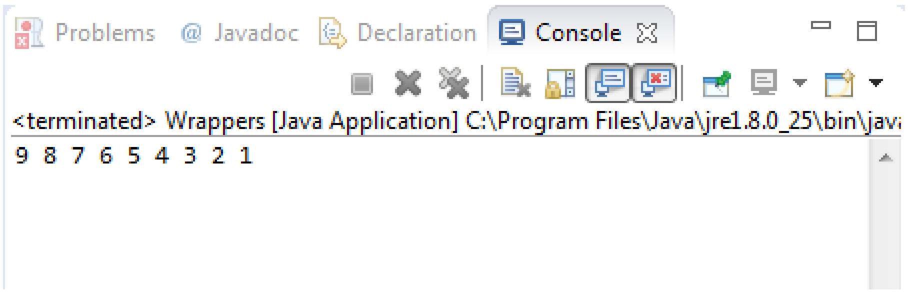
```
input:  int[] data

for (int i = 0; i < data.length / 2; i++)
{
     swap data[i] with data[data.length – (i+1)]
}
```

Write the code:

Suppose you also want to be able to reverse arrays of `doubles`, arrays of `chars`, and arrays of `Strings`. You could write three more methods that are the same except for the type of the array in the parameter list. Or you could make the method generic. Wherever you would put the type of the elements that you need to make generic, you put a capital letter that serves as a *generic type parameter*. Programmers typically use T or E for such types. You also put that capital letter inside a set of angle brackets right before the return type of the method.

Here is the generic `reverseArray` method along with a main method to illustrate that the method works (see the output below the code):

```
public class Wrappers
{

    public static <T> void reverseArray(T[] data)
    {
        for (int i = 0; i < data.length / 2; i++)
        {
            T temp = data[i];
            data[i] = data[data.length - (i + 1)];
            data[data.length - (i + 1)] = temp;
        }
    }
    public static void main(String[] args)
    {
        Integer[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        reverseArray(array);
        for (Integer n : array)
        {
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```



```
<terminated> Wrappers [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\jav
9 8 7 6 5 4 3 2 1
```

There are some rules for using generic types due to the way that the Java virtual machine is designed:

- You may not use the `new` keyword to create an object of the generic type.
- You may not create an array of the generic type.

These restrictions are due to a compilation technique called *erasure*, in which the exact data type of a generic type is unknown at run time when a generic method is running.

Generic Classes

The same way that a generic method can depend on an unspecified (at compile time) underlying data type, a generic class can do the same thing. Container classes are the perfect example. The underlying logic of the container data structure does not depend on what type of data the container holds – just as the array reversal method did not depend on what type of data was in the array.

We make a class be generic by putting a generic type parameter (such as E or T) in angle brackets immediately after the class name. Throughout the rest of the implementation, we use the type parameter as if it were any other class name.

The class header of `DoubleLinkedSeq` is shown here:

```
public class DoubleLinkedSeq implements Cloneable
```

Suppose you want to implement a linked sequence class that can hold any kind of objects. Write the class header for a `LinkedSeq` generic class:

The `DoubleNode` class will not work for our generic `LinkedSeq`. Instead, we need a generic `Node` class so that our nodes can hold any kind of data. Here is the beginning of our generic `Node` class:

```
public class Node<E>
{
   // Invariant of the Node class:
   //   1. Each node has one reference to an E Object, stored in the instance
   //      variable data.
   //   2. For the final node of a list, the link part is null.
   //      Otherwise, the  link part is a reference to the
   //      next node of the list.
   private E data;
   private Node<E> link;
```

Note that the data field of a `Node` is of whatever type a user provides to Java when the `LinkedSeq` is declared. Let's go back to the generic `LinkedSeq` class and change its fields.

This is the way they looked in DoubleLinkedSeq:

```
    private int manyNodes;
    private DoubleNode head;
    private DoubleNode tail;
    private DoubleNode cursor;
    private DoubleNode precursor;
```

What changes need to be made to these fields in our generic LinkedSeq class?

Here is the addBefore method in the DoubleLinkedSeq class. What changes need to be made to it in our generic LinkedSeq class?

```
   public void addBefore(double element)
    {
        if (manyNodes == 0)
        {
            head = new DoubleNode(element, null);
            tail = head;
            cursor = head;
        }
        else if (cursor == head || cursor == null)
        {
            head = new DoubleNode(element, head);
            cursor = head;
        }
        else
        {
            precursor.setLink(new DoubleNode(element, cursor));
            cursor = precursor.getLink();
        }
        manyNodes++;
    }
```

There are some other things that have to be understood when changing array-based container classes to generic containers. Suppose we want to make the DoubleArraySeq into a generic sequence class built on an array implementation. Here is the current class header. Change it to a generic class named ArraySeq.

```
public class DoubleArraySeq implements Cloneable
```

The fields of DoubleArraySeq are as follows:

```
private double[] data;

private int manyItems;

private int currentIndex;
```

The indexes can stay the same, but what about the array? We are not allowed to create an array that holds objects of unknown type E. Instead, we have to declare data as an array of Java `Objects`. Write the declaration of data here:

In the constructor, we make a new array of `Objects`, like this:

```
        data = new Object[DEFAULT_CAPACITY];
```

Now, whenever we need to retrieve a value from the array, we are going to have to cast it to an (E) object because we will be doing a narrowing conversion. For example, in the `getCurrent` method, we will have to say

```
        return (E) data[currentIndex];
```

Warnings in Generic Code

The Java compiler issues warnings when it compiles code with generic types to remind us that we need to be careful in order to avoid code that crashes at runtime. When you have an array of `Objects` and you typecast one of the objects to (E), Java will issue a warning that this is an "unchecked cast." Java is telling you that at runtime it will not be able to verify that the type of the object matches the type of the return statement or the type of the variable you are assigning it to (due to erasure). If you are careful in your coding, you will know that the types will match. You can suppress the Java warnings by putting the following line just before any method that contains a warning:

```
    @SuppressWarnings("unchecked")
```

Recall that we are not allowed to create arrays of a generic type. Thus, variable arity methods will generate warnings when they are of a generic type, since Java has to build an array to hold the values. You can suppress these warnings about variable arity methods with

```
    @SafeVarargs
```

above any method that declares a variable arity parameter or any method that calls another method that has a variable arity parameter list.

Using Generic Classes

A program that wants to use a generic class has to tell Java what class will be used for the generic type parameter. This is called *instantiating* the generic type parameter. How would we declare a `LinkedSeq` of `Strings`?

Converting Classes to Generic Classes

In the `SortedLinkedList` lab, you will convert a class to generic. Take your book to lab so that you can refer to the steps listed on pages 269 – 270. Be sure you understand the significance of each step. Here is a brief version of the rules:

1. Change the name of the class and append <E> or <T>. Do not put the type parameter on the names of the constructors.
2. Change any data arrays to arrays of `Objects`.
3. Wherever the class referred to the original data type of the elements in the container, change the type to E or T.
4. Change static methods to generic static methods.
5. Use a typecast when you retrieve an element from an `Object` array.
6. Suppress warnings.
7. If you used == to compare values from the container to each other or to a parameter, change statements to use equals instead.
8. A generic container stores references. You need to decide whether or not null is a value you might store in the container. Some methods will need special cases to deal with such values because you use == to compare to null, but you use equals to compare to objects.
9. Set unused references in a partially filled array to null.
10. Update all the documentation.

For practice, change this class to a generic class:

```
public class IntLinkedBag implements Cloneable
{
   private IntNode head;
   private int manyNodes;


   public IntLinkedBag( )
   {
      head = null;
      manyNodes = 0;
   }
```

```
public void add(int element)
{
    head = new IntNode(element, head);
    manyNodes++;
}

public void addAll(IntLinkedBag addend)
{
    IntNode[ ] copyInfo;

    if (addend.manyNodes > 0)
    {
        copyInfo = IntNode.listCopyWithTail(addend.head);
        copyInfo[1].setLink(head);
        head = copyInfo[0];
        manyNodes += addend.manyNodes;
    }
}


public void addMany(int... elements)
{
    for (int i : elements)
        add(i);
}


public Object clone( )
{
    IntLinkedBag answer;

    try
    {
        answer = (IntLinkedBag) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {
        throw new RuntimeException
        ("This class does not implement Cloneable");
    }

    answer.head = IntNode.listCopy(head);

    return answer;
}
```

```java
public int countOccurrences(int target)
{
    int answer;
    IntNode cursor;

    answer = 0;
    cursor = IntNode.listSearch(head, target);
    while (cursor != null)
    {
        answer++;
        cursor = cursor.getLink( );
        cursor = IntNode.listSearch(cursor, target);
    }
    return answer;
}


public int grab( )
{
    int i;
    IntNode cursor;

    if (manyNodes == 0)
        throw new IllegalStateException("Bag size is zero");

    i =  (int)(Math.random( ) * manyNodes) + 1;
    cursor = IntNode.listPosition(head, i);
    return cursor.getData( );
}


public boolean remove(int target)
{
    IntNode targetNode;

    targetNode = IntNode.listSearch(head, target);
    if (targetNode == null)
        return false;
    else
    {
        targetNode.setData(head.getData( ));
        head = head.getLink( );
        manyNodes--;
        return true;
    }
}
```

```
   public int size( )
    {
        return manyNodes;
    }


    public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)
    {
        IntLinkedBag answer = new IntLinkedBag( );

        answer.addAll(b1);
        answer.addAll(b2);
        return answer;
    }

}
```

Java Wildcards

There is a hierarchy in the Java API for all the generic collection classes, with a generic interface named `Collection` at the top. `ArrayList`, `Vector`, `LinkedList`, and many other collection classes implement `Collection`.

Consider the problem of writing a routine that prints all the elements in some collection, regardless of the type of the element. You might think this would work:

```
public static void printCollection(Collection<Object> c)
{
    for (Object e : c)
    {
        System.out.println(e);
    }
}
```

but it wouldn't. That method would only work on collections of `Objects`, not collections of subclasses of `Object`. `Collection<Object>` is not a superclass of `Collection<Integer>` or `Collection<String>`. What is the supertype of all kinds of `Collections`? It is `Collection<?>`, read "collection of unknown." The ? is called a wildcard type. The above method, with `Collection<?>` c in the parameter list, can be called with any type of collection.

```
public static void printCollection(Collection<?> c)
{
    for (Object e : c)
    {
        System.out.println(e);
    }
}
```

Inside `printCollection`, you can safely assign elements from the collection of unknown to type `Object` since elements of any type are `Objects`. But you could not write a method that took a parameter with an unknown type and add arbitrary new elements to it. This would produce a compiler error.

Suppose you wanted to write a sort method that receives an array of elements to be put in order. The code for sorting `ints`, `doubles`, and `Strings` should be about the same. It seems like a good time to use generics, but there are difficulties. To sort, we have to be able to compare the things we are sorting. If we were comparing `ints` or `doubles`, we could use the less than or greater than symbols, but with generics, we will be working with objects. Objects often are compared using the `compareTo` method, as described in the textbook. The problem is that not all classes implement the `compareTo` method, and if our method is called with an array of some type that does not have a `compareTo`, then our code won’t work.

Java allows a special syntax that allows us to specify that the generic must implement a certain interface:

```
        public static <T extends Comparable <? super T>> sort (T[] arrayToSort)
        {
             code for our sort
        }
```

The strange underlined section above is saying that the sort method is a generic method, the generic type is T, and that whatever type we substitute for T must either implement the `Comparable<T>` interface itself, or a superclass of T implements the `Comparable<T>` interface. For example, if the `Student` class implemented `Comparable<Student>` and `GraduateStudent` was a subclass of `Student`, then `GraduateStudent` objects could be sorted as well as `Student` objects, even if the `GraduateStudent` class did not have a `compareTo` method.

You can also write generic classes that have a similar requirement:

```
    public class SortedList<E extends Comparable<? super E>>
```

Explain what that syntax is saying:

Iterators (Section 5.5)

Recall from CS 1440 how an iterator was used to visit each item in a collection:

```
ArrayList<String> names;
// code that puts Strings in the ArrayList…

Iterator<String> itr = names.iterator();
while (itr.hasNext())
{
    System.out.println (itr.next());
}
```

1. How does the code above "get" the new `Iterator<String>`?

2. What is wrong with the following code?

```
int countShortNames = 0;
int countLongNames = 0;
Iterator<String> itr = names.iterator();
while (itr.hasNext())
{
    if (itr.next().length() < 5)
        countShortNames++;
    else if (itr.next().length() > 10)
        countLongNames++;
}
System.out.println ("There were " + countShortNames + " short names and " +
                    countLongNames + " long names.");
```

3. If a collection implements the __________________ interface, then a programmer can use a for/each loop to iterate over the collection.

4. What one method is required to implement the interface above? Give the signature for the method. How does that method do its job?

5. `Iterator<E>` is an interface – not a class. How can a method return an `Iterator<E>`?

6. The textbook gave an example of an `Iterator` for its `LinkedBag` class that was called `Lister`. How did the generic bag class implement the iterator method from the `Iterable` interface? See page 298. Note that their cast is not required.

7. What is the difference between the `Iterable<E>` interface and the `Iterator<E>` interface?

8. What methods are in the `Iterator<E>` interface?

9. The textbook created a class (`Lister`) that implemented the `Iterator` for the `LinkedBag` as a separate class from the `Bag`. A more common technique is to create an inner class – a class inside another class. The only class that can create objects of this inner class is the outer class. Look at the example of part of a possible implementation of part of an `ArrayList` class on the last page of this handout. What is the name of the inner class?

10. Which class implements the `Iterable` interface?

11. Which class implements the `Iterator` interface?

12. What is the `remove` method supposed to do? (You are never allowed to do two `removes` without a `next` between them.)

13. Imagine that the array in an `ArrayList` contains `Integers` and is as follows. Trace how the iterator in the code on the last page of this handout executes the following code. Take note of how the `lastReturnedIndex` field is used to make sure that a remove is always preceded by a call to next.

55	11	12	17	82	64

```
Iterator<Integer> it = list.iterator();
while (it.hasNext())
{
      Integer x = it.next();
      if (x % 2 == 0)
      {
            it.remove();

      }
}
```

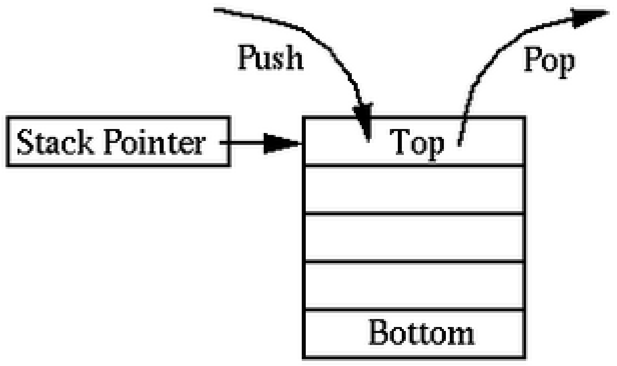
14. Implementing a `remove` method is the biggest hassle for a class implementing an `Iterator`. It is optional to do a true implementation of a `remove` in a class implementing an iterator. What can the `remove` do instead?

15. In lab, you will be taking a `DoubleSortedLinkedList` class and (1) changing it to a generic `SortedLinkedList` class, and (2) adding an `Iterator` to the class. There will be extra credit for students who support a true `remove` operation. Can you think of what extra fields or work would be needed to support a `remove` operation?

```
public class ArrayList<E> implements Iterable<E>
{
    private int manyItems;
    private Object[] data;
    public ArrayList ()
    {
        manyItems = 0;
        data = new Object[10];
    }
    public Iterator<E> iterator()
    {
        return new ArrListIter();
    }
    // … other ArrayList methods
    private class ArrListIter implements Iterator<E>
    {
        int index;                 // is equal to manyItems when no current item
        int lastReturnedIndex;    // -1 if next has not been called since last remove

        private ArrListIter()
        {
            index = 0;
            lastReturnedIndex = -1;
        }
        public boolean hasNext()
        {
            return (index < manyItems);
        }
        @SuppressWarnings("unchecked")
        public E next()
        {
            if (!(hasNext()))
            {
                throw new NoSuchElementException ("No more items to iterate.");
            }
            lastReturnedIndex = index;
            index++;
            return (E) data[lastReturnedIndex];
        }
        public void remove()
        {
            if (lastReturnedIndex == -1)
            {
                throw new IllegalStateException("Remove called before next");
            }
            for (int i = lastReturnedIndex; i < manyItems-1; i++)
            {
                data[i] = data[i+1];
            }
            data[manyItems-1] = null;
            manyItems--;
            index = lastReturnedIndex;
            lastReturnedIndex = -1;
        }
    }
```

Stacks (Sections 6.1, 6.2)

A stack is a data structure of ordered items such that items can be inserted and removed only at one end (called the **top**). Stacks are often given the acronym LIFO (last in, first out).



A LIFO Stack

The specification of a generic `Stack` includes the following methods:

- `public boolean isEmpty()`
- `public E peek()`
- `public E pop()`
- `public void push(E item)`
- `public int size()`

If a user attempts to peek or pop an empty stack, it results in an error condition called *stack underflow*. There is an exception in Java called `EmptyStackException` that we will throw from our implementations of `peek` and `pop`.

The most interesting thing about stacks is how useful they are! We will study a few applications in this course but you will see many others in your other courses. First, let's do some exercises to be sure that you understand how stacks work.

1. List the elements in the order in which they are popped from the stack, given this series of stack operations: `push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop(), pop().`

2. Consider a series of pushes and pops in which you pushed the integers 1, 2, 3 (in that order). Could you have popped them off in the order 1,2,3? What about 3,2,1? What ordering of the numbers 1,2,3 would be impossible as the popped order?

3. If you pushed the four number 1, 2, 3, 4, in that order, name some orderings which would be impossible as the pop order.

4. Suppose a stack has the values {3, 4, 8, 12, 15, 12, 3, 4, 5, 4}, in that order, where 4 is at the top of the stack. Trace the following method on that stack, and tell what the stack would look like when the method ends.

```java
public static <T> void traceThis(Stack<T> stack)
{
    ArrayList<T> list = new ArrayList<T>();
    T temp;
    while (!stack.isEmpty())
    {
        temp = stack.pop();
        if (!(list.contains(temp)))
        {
            list.add(temp);
        }
    }
    for (int i = list.size() -1; i >= 0; i--)
    {
        stack.push(list.get(i));
    }
}
```

5. Suppose a stack, `myStack`, contained the values {5, 3, 1, 2, 4}, with 4 at the top. What would be the result of the statement, `myStack = popAndPush(myStack),` given the following method?

```
public static <T> Stack<T> popAndPush(Stack<T> stack)
{
    Stack<T> newStack = new Stack<T>();
    while (!stack.isEmpty())
    {
        newStack.push(stack.pop());
    }
    return newStack;
}
```

6. Here is a very useful and clever method. Trace its behavior on the following inputs and say what it returns.

```
public static String mystery(int num, int b)
{
    String digitChar = "0123456789ABCDEF";
    Stack<Character> stack = new Stack<Character>();
    String answer = "";
    do
    {
        stack.push(digitChar.charAt(num % b));
        num /= b;
    } while (num != 0);

    while (!stack.isEmpty())
        answer += stack.pop();
    return answer;
}
```

a. mystery(531, 7)

b. mystery(20, 2)

c. mystery(173, 16)

7. In section 6.2, the author gave us the following algorithm for evaluating a fully parenthesized infix expression:

```
initialize a stack of characters
initialize a stack of doubles
while there is more of the expression to read
{
    if the next item in the expression is a number
    {
         read it and push it on the stack of doubles
    }
    else
    {
         read the next char and put it in the variable, symbol
         switch on symbol
         {
              case symbol is an operator:
                   push symbol on the stack of characters – break
              case symbol is ')':
                   call evaluateStackTops and pass the two stacks – break
              case symbol is '(' : don't do anything – break
              default : throw an IllegalArgumentException
          }

     }
}
if the stack of doubles doesn't have just one thing in it
{
     throw an IllegalArgumentException
}
pop and return the value from the stack of doubles
```

`EvaluateStackTops` gets an operator off the stack of characters and gets the top two values off the stack of doubles. It applies the operator to the values and pushes the result back on the stack of doubles.

Carry out the operations of the algorithm on the following two expressions:

a. ((12 * 3) + ((20 * 2) 5))

b. ((12 + (((6 * 3) – 10) / 4)) * (5 + 2))

Implementations of Stacks (Section 6.3)

Array Implementation

Suppose we decide to implement our stack specification with an array. We want to avoid ever having to shift elements in the array. Where is the best place to put the top of the stack?

Suppose our fields are a `data` array and a count of elements called `manyItems`.

- `private Object[] data`
- `private int manyItems`

The invariant for our design is as follows:

1. The number of items in the stack is stored in the instance variable, `manyItems`.
2. The items in the stack are stored in a partially filled array called `data`, with the bottom of the stack at `data[0]`, the next item at `data[1]`, and so on, to the top of the stack at `data[manyItems-1]`.

Describe how each of the following methods is implemented and give its big-oh complexity:

- `public ArrayStack()`

- `public ArrayStack(int initialCapacity)`

- `public ArrayStack<E> clone()`

- `public void ensureCapacity(int minimumCapacity)`

- `public void trimToSize()`

- `public int getCapacity()`

- `public boolean isEmpty()`

- `public E peek()`

- `public E pop()`

- `public void push(E item)`

- `public int size()`

Linked List Implementation

A linked list is a very suitable way to implement a stack since the list can grow and shrink easily, and the head of the linked list is an efficient place to push and pop. Not using an array eliminates the need for the methods `ensureCapacity`, `trimToSize`, and `getCapacity`.

The instance variables of our linked list implementation are:

- `private Node<E> top`
- `private int manyItems`

Note that the author chose not to keep the counter, `manyItems`. Instead, he calls `Node.listLength(top)` in the size method. This is an unwise choice in your instructor's opinion, so we will keep the counter.

Describe how each of the following methods is implemented and give its big-oh complexity:

- `public LinkedStack()`

- `public LinkedStack<E> clone()`

- `public boolean isEmpty()`

- `public E peek()`

- `public E pop()`

- `public void push(E item)`

- `public int size()`

Stack Applications (Section 6.4)

Infix, Postfix, and Prefix
Infix, postfix and prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

- Infix notation: X + Y
  Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

  Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

- Postfix notation (also known as "Reverse Polish notation"): X Y +
  Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /

  The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

  Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:

  ((A (B C +) *) D /)

  Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

- Prefix notation (also known as "Polish notation"): + X Y
  Operators are written before their operands. The expressions given above are equivalent to / * A + B C D

  As for postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

  (/ (* A (+ B C)) D)

Although prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication). Because postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in prefix expressions.

Write the following completely parenthesized expressions in prefix and in postfix notation.

- ((4 + 5) * (3 – (8 / 4)))

  prefix ____

  postfix ____

- ((8 – (5 * 2)) + (4 * 6))

  prefix ____

  postfix ____

Evaluating Postfix
Here is the author's procedure for evaluating a postfix expression:

```
Initialize a stack of double numbers.
While there is more of the expression to read
    if (the next input is a number)
        Read the next input and push it onto the stack.
    else
    {
        Read the next character, which is an operation symbol.
        Pop two numbers off the stack.
        Combine the two numbers with the operation
            (using the second number popped as the left operand)
            and push the result onto the stack.
    }
At this point, the stack contains one number, which is the value of the
expression.
```

Use the algorithm to evaluate the following postfix expressions:

- 12 3 4 + * 2 / 5 – Answer ____________________
- 10 9 8 7 5 – / + * Answer ____________________
- 14 15 + 12 – 3 * 17 / Answer ____________________

In any arithmetic expression, what is the proportion of operands to operators?

Converting Infix Expressions to Postfix

Here is the author's algorithm for converting infix expressions to postfix. The infix expression does not need to be fully parenthesized. Note that this is an easier to understand piece of pseudocode than the one in the text, so use this one when you implement your second calculator in the lab.

```
initialize a stack of characters to hold the operation symbols and parentheses
while there is more of the expression to read
{
    if (the next input is a number)
        read the number and write it to the output followed by space
    else
    {
        read the next symbol
        if the symbol is '('
        {
            push it on the stack.
        }
        else if the symbol is an operator
        {
            while the stack is not empty and
                  the char at the top of the stack is not '('
                  (it is an operator), and
                  the operator at the top of the stack has equal or higher
                  precedence than the symbol
            {
                pop the operator off the stack and write it to the output
            }
            push the symbol onto the stack.
        }
        else if the symbol is ')'
        {
            while the stack is not empty and
                  the character at the top of the stack is not '('
            {
                pop the operator off the stack and write it to the output
            }
            pop the stack and discard the '('
        }
        else there is an error
    }
}
while the stack is not empty
{
        pop the stack and write it to the output
}
```

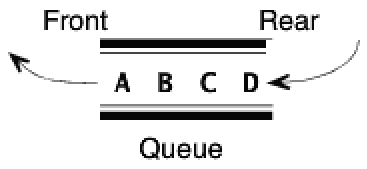
Convert the following to postfix using the above algorithm. Show the stack, putting a cross through the values as they are popped.

18 + 30 / (12 + 3) – 4 Answer: ____________________

(20 – (1 + 6) * 2) / (1 + 2) Answer: _____________________

Queues (Sections 7.1, 7.2)

A queue is a data structure of ordered items such that items can be inserted only at one end (called the *rear*) and removed at the other end (called the *front*). A queue is often tagged with the acronym FIFO, first-in first-out. Queues are orderings that people are familiar with from their everyday lives. We line up in queues to wait for our turn to get food in a cafeteria, buy tickets at the theater, pay for groceries, and so forth. We get in line at the end (the rear) and we get to be waited on when we reach the front.



Typically, the specification of a generic Queue includes the following methods:

- `public boolean isEmpty()`
- `public E peek() (sometimes called examine)`
- `public E remove() (sometimes called dequeue or poll)`
- `public void add(E item) (sometimes called enqueue or insert)`
- `public int size()`

Unlike the case with `Stack<E>`, there is no `Queue<E>` class in the Java library. There is a `Queue<E>` interface, however, that is implemented by several Java classes, `LinkedList` among them. Programmers who need to declare a `Queue` typically use the Java `LinkedList` class as shown here:

```
Queue<Integer> q = new LinkedList<Integer>();
```

When we discuss the implementations of queues, we'll see why a linked list is a very effective data structure for a queue.

While `Stacks` reverse a series of elements, `Queues` keep them in the same order. What does this method accomplish?

```
public static <T> void mystery(Queue<T> q)
{
    Stack<T>  s = new Stack<T>();
    T element;
    while(!q.isEmpty()) {
        element = q.dequeue();
        s.push(element);
    }
    while (!s.isEmpty()) {
        element = s.pop();
        q.enqueue(element);
    }
}
```

Here is an interesting method. Study the code and then answer the following questions.

```
public static void main(String[] args)
{
    Queue<Integer> q = new LinkedList<Integer>();
    Scanner keyIn = new Scanner(System.in);
    for (int i = 1; i <= 5; i++)
    {
        if (keyIn.nextBoolean())
            System.out.print (i + " ");
        else
            q.add(i);
    }
    while (!q.isEmpty())
    {
        System.out.print (q.remove() + " ");
    }
}
```

If the user types “true false false true true,” what will the code output?

Is it possible to get the output “1 3 5 4 2”? If so, give an input sequence that produces the output. If not, explain why not.

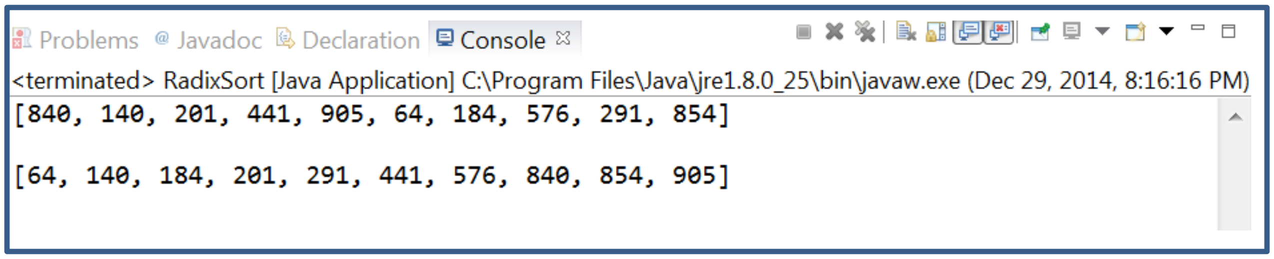
List all the inputs that would result in the output “1 2 3 4 5.”

Radix Sort

A radix sort can be used to sort data such as telephone numbers, zip codes, product id's, and so forth – data that is fixed in length and composed of a known set of characters and/or digits. Radix sort is implemented with an array of queues, each of which is called a bucket or a bin. It will better enable us to describe the steps of the algorithm if we describe the input mathematically, as follows.

Suppose we are sorting a list of fixed-length integers. Suppose each integer is composed of m digits. We can think of each integer as having the form $d_{m-1}d_{m-2} \ldots d_2 d_1 d_0$, where d_{m-1} is the most significant digit and d_0 is the least significant.

The program on the next page generates 10 3-digit numbers randomly and inserts them into a queue. It then passes that queue to the radix sort method, which sorts the queue and returns it. You can see a sample run of the program here:



```
<terminated> RadixSort [Java Application] C:\Program Files\Java\jre1.8.0_25\bin\javaw.exe (Dec 29, 2014, 8:16:16 PM)
[840, 140, 201, 441, 905, 64, 184, 576, 291, 854]

[64, 140, 184, 201, 291, 441, 576, 840, 854, 905]
```

Your instructor will show you the 3 stages of a radix sort on the first (unsorted) list of 3-digit numbers:

```java
import java.util.Queue;
import java.util.LinkedList;

public class RadixSort
{
    public static void main(String[] args)
    {
        Queue<Integer> data = new LinkedList<Integer>();
        for (int i = 0; i < 10; i++)
        {
            data.add((int) (Math.random() * 999) + 1);
        }
        System.out.println(data);
        System.out.println();

        data = radixSort(data);
        System.out.println(data);
    }

    public static Queue<Integer> radixSort(Queue<Integer> data)
    {
        // make the buckets
        Queue<Integer>[] bucket = new LinkedList[10];
        for (int i = 0; i < 10; i++)
            bucket[i] = new LinkedList<Integer>();
        int index;
        int x;
        int placeValue = 1;  // start in the 1's column

        for (int stage = 0; stage < 3; stage++)  // for each digit
        {
            // distribute the values into the buckets
            while (data.size() > 0)
            {
                x = data.remove();
                index = x / placeValue % 10;
                bucket[index].add(x);

            }
            // empty the buckets and reassemble the queue
            for (int i = 0; i < 10; i++)
            {
                while (bucket[i].size() > 0)
                {
                    data.add(bucket[i].remove());

                }
            }
            placeValue *= 10;  // go to the next column
        }
        return data;
    }
}
```

Now suppose we are sorting 15-letter strings with each letter in the range [a – z] using radix sort.

How many stages would there be in the algorithm?

How many buckets would be needed?

How could we index into an array, based on a letter of the alphabet?

What if some of the strings were shorter than 15 characters? How could we handle the mixture of strings of varying lengths?

The complexity of a radix sort is O(m(n + b)), where b represents the number of buckets, n is the number of data items to be sorted, and m is the length of each data item.

What is the complexity of sorting n social security numbers?

Why might radix sort be called a linear sort?

Java syntax note:

Arrays of `Stacks` or `Queues` or `LinkedLists` are very useful. Java gives compile time errors when we try to create arrays of generic types. Did you notice the line in the above program that looks like this?

```
        Queue<Integer>[] bucket = new LinkedList[10];
```

Compare it to this line:

```
            bucket[i] = new LinkedList<Integer>();
```

To avoid syntax errors, do not use the generic type when you make a new array of some generic container type.

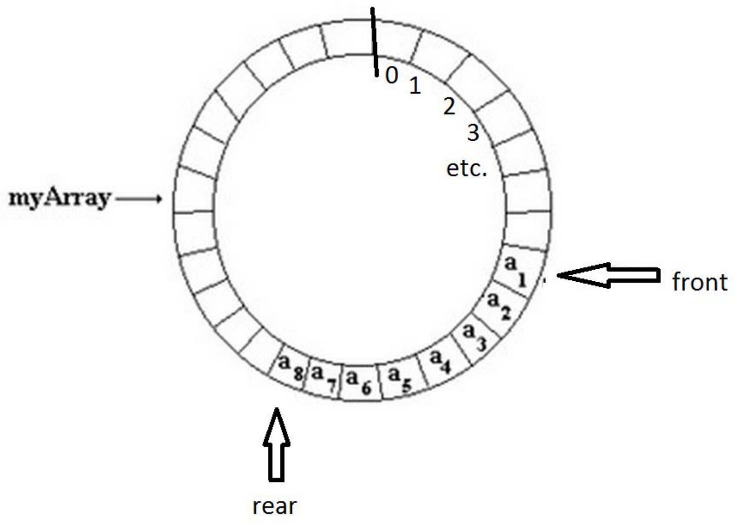
Queue Implementations (Section 7.3)

We will examine both an array implementation of a queue and a linked list implementation.

Array Implementation – Circular Array

We want to have an (amortized) complexity of O(1) for add and remove. Therefore, we can't do any shifting of the elements in the array. The only time we will tolerate an O(n) operation is when we have to grow the array prior to completing an add, and if that happens only rarely, we can amortize it away. We can achieve our efficient adds and removes with a circular array.

We keep two indexes, `front` and `rear`. Every time we add an element, we increment the `rear` index. Similarly, every time we remove an element, we increment the `front` index. Over time the section of the array that contains the elements of the queue moves toward the end of the array. To reuse the unused space at the beginning of the array, we think of the array as circular. When an index variable (`front` or `rear`) reaches the end of the array, we wrap it back around to the beginning.



To reiterate, what are the fields for the `ArrayQueue`?

Suppose a queue with capacity 10 is being used, and at the present time it contains the `Integers` 6, 8, 4, 2, and 14, added in that order. Further suppose that `manyItems` = 5, `rear` = 2, and `front` = 8. Draw the corresponding array.

Suppose a queue has been implemented as an array as described above. Draw the `data` array after the following operations have been performed.

```
String[] words = {"one", "two", "three", "four", "five", "six"};
ArrayQueue<String> queue = new ArrayQueue<String>();
for (int i = 0; i < words.length; i++)
{
    queue.add(words[i]);
    if (i % 3 == 2)
        queue.remove();
}
queue.remove();
queue.remove();
for (int i = words.length-1;  i >=0; i--)
    queue.add(words[i]);
```

The author uses a private helper method called `nextIndex` to increment both `front` and `rear`. He uses the method as follows:

```
front = nextIndex(front);
```

This method adds 1 to the index it receives and returns that new value unless the new value has gone off the end of the array. In that case, the method returns 0. There are various ways to implement the `nextIndex` method. Let's write three.

Linked List Implementation

For the linked list implementation of a queue there are three instance variables:

- `int manyNodes`
- `Node<E> front`
- `Node<E> rear`

It is efficient to both add and remove nodes from the head end of a singly-linked list. It is efficient to add nodes to the tail end. However, it is not efficient to remove nodes from the tail. Why not?

With this in mind, which end of a linked list should we use for the front of the queue and which end should we use for the rear?

Describe the implementations of each of the following methods in a `LinkedQueue` implementation.

- `public void add(E element)`
- `public E remove()`
- `public int size()`
- `public Boolean isEmpty()`
- `public E peek()`

Fill in the following table, showing the big-Oh evaluation for the method with the given implementation.

	ArrayQueue	**LinkedQueue**
add(E element)		
remove()		
isEmpty()		
size()		
peek()		

A deque (pronounced "deck") is a structure in which you can both add and remove from either the front or the rear. Suppose you wanted to change the `ArrayQueue` or the `LinkedQueue` to an `ArrayDeque` or a `LinkedDeque`. Comment on the way you would implement each of the following operations:

a. adding to the front of an array deque

b. removing from the rear of an array deque

c. adding to the front of a linked list deque

d. removing from the rear of a linked list deque

Exam 2 Study Guide

Chapters covered: 4 (Linked Lists), 5 (Generics), 6 (Stacks), 7 (Queues)
Remember that you can have a reference sheet, so make yourself a good one.

1. Know how to get random integers in any range.
2. Understand the IntNode class from the text (and other such Node classes, including the generic Node class). Know what methods the class has and how those methods do their jobs. Be able to use those methods to insert nodes into a linked list, remove nodes from a linked list, copy all or part of a linked list, and so forth. Understand the big-oh complexity measurements of each of the methods.
3. Know how the IntLinkedBag class works and how it was implemented.
4. Know how the DoubleLinkedSeq class works and how it was implemented.
5. Know the advantages and disadvantages of array and linked list implementations of container classes.
6. Understand the Java wrapper classes.
7. Know how to write generic methods.
8. Know how to convert a container class into a generic class.
9. Understand how to use an iterator and how to create one. Be able to explain when to implement the Iterator interface and when to implement the Iterable interface. Know the methods of each of those interfaces.
10. Know how stacks work and how to implement them both with arrays and with linked lists.
11. Know how to convert infix to postfix, both by hand and with code.
12. Know how to evaluate a postfix expression.
13. Know how queues work and how to implement them both with circular arrays and with linked lists.
14. Know how radix sort works and how to run it on a list of *k*-digit integers.

Recursion (Sections 8.1, 8.3)

From your reading of chapter 8, answer the following questions.

1. What is a recursive method?

2. When should you consider using recursion?

3. A recursive algorithm should always have a base case. What is meant by a base case?

4. A recursive call should always be with a simpler instance of the problem, i.e., the recursive call should always be closer to a base case. Give an example of a recursive call from your reading, and tell in what way the call was simpler.

5. What is wrong with each of the following attempts to write a recursive method? Can you fix each?

```java
// prints a linked list of Strings in reverse
public void printInReverse(Node<String> r)
{
    printInReverse(r.getLink());
    System.out.println(r.getData());
}

// Fibonacci Numbers
// fib(0) = 1, fib(1) = 1
// otherwise,  fib(n+1) = fib(n) + fib(n-1)
public static int fib (int n)
{
    if (n < 2)
        return 1;
    else
        return fib(n+1) - fib(n-1);
}
```

6. Study this method and then answer the questions below.

```
public String seenThisBefore(int n, int b)
{
    String digit = "0123456789ABCDEF";
    if (b > n)
        return "" + digit.charAt(n);
    else
        return "" + seenThisBefore(n / b, b) + digit.charAt(n % b);
}
```

a. What is the base case of the method?

b. What does the method return when you call it with n = 10 and b = 2?

c. What does the method do?

d. When did you see an equivalent method before?

7. What is meant by an execution stack or a run-time stack? What is meant by an activation record?

8. What does this method do if it is called with 825?

```
public static void bedtimeStory (int time)
{
    if (time >= 830)
        System.out.println("OK, brats. Get to bed!");
    else
    {
        System.out.println("It was a dark and stormy night,");
        System.out.println("and a bunch of spoiled children were");
        System.out.println("trying to avoid bedtime.");
        System.out.println("One of them said, 'Dad tell us a story,'");
        System.out.println("and Dad said: ");
        bedtimeStory(time+1);
        System.out.println("THE END!");
    }
}
```

9. Tracing recursive methods is the beginning of a full understanding of recursion. Writing your own recursive methods is the next step. Let's write a recursive method, `numToNames`, that converts a positive integer to a `String` with word names for the digits. For example, `numToNames`(372) returns the `String` "three seven two."

What should be the base case?

What should be the recursive call? Be sure that it makes progress toward the base case.

Now put the method together.

10. Write a recursive method that finds the sum of the digits of a non-negative integer. You can use the same base case and a similar recursive call to what you used previously before.

11. Sometimes when you are working with recursion it is useful to send extra parameters that specify the subtask. For example, here is a recursive method for finding the sum of an array. The method finds the sum from a starting index to the end.

```
public int sumArray(int index, int[] array)
{
    if (index >= array.length)
        return 0;
    else
        return array[index] + sumArray(index+1, array);
}
```

Usually a non-recursive version of the method calls the recursive method with the initial parameters.

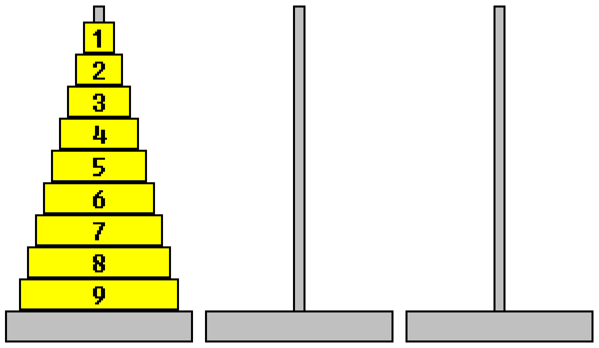
```
public int sumArray(int[] array)
{
    return sumArray(0, array);
}
```

Write a recursive method, isPal, that determines whether a String is a simple palindrome made of letters [a – z]. Palindromes are strings like "dad", "mom", "toot", and "madamimadam," i.e., strings that are spelled the same way forward and backward. The recursive method will take a `String` and a left and right index as arguments. Also write a non-recursive method that receives just the `String` and makes an appropriate call to the recursive method.

```
public boolean isPal(String str, int left, int right)
```

```
public boolean isPal(String str)
```

12. The towers of Hanoi is a famous puzzle with a recursive solution. The puzzle consists of three poles and a collection of rings that stack on the poles. The rings are different sizes and are stacked in decreasing order of their size on the first pole. The second and third poles are empty. The goal is to move the rings from the first pole to the second pole, one at a time, under the restriction that you cannot put a larger ring on top of a smaller one. You can use the third pole to hold rings temporarily.



Your instructor will show you an algorithm that you can use to solve the puzzle by hand.

This recursive method will print out instructions for solving the puzzle. Trace its behavior on the calls solveIt(2, "1", "2", "3") and solveIt(3, "1", "2", "3").

```
public void solveIt(int numRings, String start,  String end,
    String temp)
{
    if (numRings == 1)
        System.out.println ("From " + start + " to " + end);
    else
    {
        solveIt(numRings-1, start, temp, end);
        System.out.println ("From " + start + " to " + end);
        solveIt(numRings-1, temp, end, start);
    }
}
```

Figure out how many moves are needed for the following numbers of rings.

Number of Rings	Number of Moves
1	
2	
3	
4	
n	

Sequential and Binary Search (Section 11.1)

Sequential Search

Sequential search is the technique you have to use when the data you're searching through is in no particular order. Think about a box of baseball cards that are not organized in any way. In order to find your favorite Mickey Mantle card, you'll have to start at the beginning and look through all the cards until you either find it or you have examined the entire box and not located the card.

Write a sequential search method, given the following description and method signature:

```
/** Search a portion of an array for a target
 * @param  arr   the array to be searched
 * @param  first  the index to start the search
 * @param  size   the number of elements to be searched
 * @param  target the element to search for
 * @return the index of the first occurrence of the target or -1 if not there
 */
public static int search (Object [] arr, int first, int size, Object target)
```

What is the big-oh complexity of an unsuccessful sequential search? ________________

What is the big-oh complexity of a successful sequential search? ____________________

There are other common method signatures for a sequential search:

-
  ```
  // search the entire array for the first occurrence of target
  public static int search(Object[] array, Object target)
  ```

-
  ```
  // search between firstIndex (inclusive) and lastIndex (inclusive)
  public static int search(Object[] array, int firstIndex, int
  lastIndex, Object target)
  ```

-
  ```
  // search between firstIndex (inclusive) and lastIndex (exclusive)
  public static int search(Object[] array, int firstIndex, int
  lastIndex, Object target)
  ```

The searches in the textbook use arrays of ints. You can use the comparison operators (==, <=, <, >=, >, !=) with ints. If you are going to perform a sequential search on an array of objects from some class, you should make sure that the class has a _________________ method.

Binary Search

If you have sorted your set of baseball cards, you will have a much easier time finding a particular card. Binary search is a method that works roughly like this on a sorted set of baseball cards:

1. If your unsearched set is empty, declare failure. If there is at least one card left in the unsearched portion of your set, see if what you're looking for is the "middle" card.
2. If it is, you've found it.
3. If it is not, and the middle card comes after the one you're looking for, restrict your search to the left half of the set and go to step 1.
4. If it is not, and the middle card comes before the one you're looking for, restrict your search to the right half of the set and go to step 1.

Binary search on a computer is best done in arrays. Here is the textbook's recursive binary search using generics:

```
public static <T extends Comparable<? super T>> int binarySearch
              (T[] arr, int first, int size, T target)
{
    if (size <= 0)
        return -1;
    int middle = first + size/2;
    int value = target.compareTo(arr[middle]);
    if (value == 0)
    return middle;
    if (value < 0)
        return binarySearch(arr, first, size/2, target);
    return binarySearch (arr, middle+1, (size-1)/2, target);
}
```

What are the two base cases that the algorithm checks?

If the algorithm is called with an even size, is the middle element the last element in the first half, or the first element in the second half?

If the size is an even integer, how many elements are before the middle element?

If the size is an even integer, how many elements are after the middle element?

If the size is an odd integer, how many elements are before the middle element?

If the size is an odd integer, how many elements are after the middle element?

Consider the following array. Finish filling in the second row to show how many array accesses are needed to find an element at that index. For example, if we searched for 414, we would first look at index 7, then 11, then 13, and finally find it at index 12. That would be 4 accesses.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elements	-20	4	12	22	41	42	45	48	78	90	111	403	414	508	902
Accesses								1				2	4	3	

On a sorted array with n elements, what is the highest number of accesses needed for a successful binary search?

On a sorted array with n elements, what is the highest number of accesses needed for an unsuccessful binary search?

What is the complexity of binary search?

Suppose a sorted array had 8 billion elements (more than the current population of the world). How many array accesses, worst case, would be necessary to find a target element using a binary search?

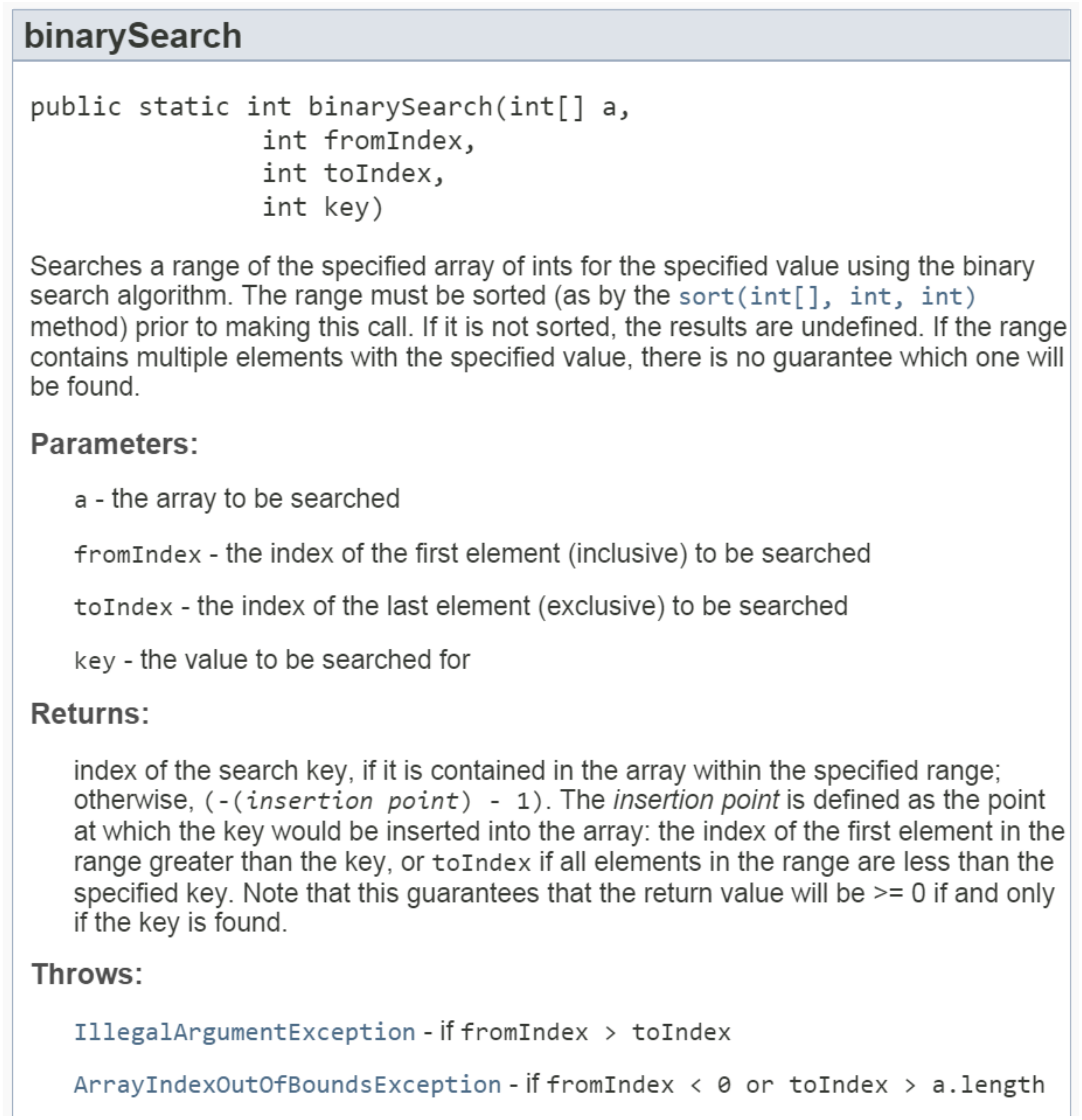
Suppose the banner id's of the 48 students in a course are stored in an unsorted array. Using a sequential search, how many array accesses, worst case, would be necessary to find a target id in the array?

The Arrays class in the Java library has many versions of binary search. There are versions with primitives and versions with objects. They tend to be in one of the following two forms:

-
  ```
  //search the entire array for the first occurrence of key
  public static int binarySearch(Object[] arr, Object key)
  ```

-
  ```
  //search between fromIndex (inclusive) and toIndex(exclusive)
  public static int binarySearch(Object[] arr, int fromIndex, int
  toIndex, Object key)
  ```

Here is the section of the API for the second version:



binarySearch

```
public static int binarySearch(int[] a,
                   int fromIndex,
                   int toIndex,
                   int key)
```

Searches a range of the specified array of ints for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(int[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

What value does the Arrays binarySearch return for an unsuccessful search?

Give the value returned for binarySearch(arr, 0, 15, 88) using the sorted array on the previous page.

The return value is supposed to give some information about where the value would fit in the array to maintain sortedness. It would seem more intuitive to return -9, since 88 would go into position 9 in the array. Why doesn't the method return -9?

Not all versions of binary search are recursive. Study this implementation:

```
public static <T extends Comparable<? super T>> int binarySearch (T[] arr,
    T target)
{
    int lowIndex = 0;
    int highIndex = arr.length - 1;
    int middleIndex, value;
    // search between lowIndex and highIndex, inclusive
    while (lowIndex <= highIndex) {
        middleIndex = (lowIndex + highIndex)/2;
        value = target.compareTo(arr[middle]);
        if (value == 0)          // target equals the middle element
            return middleIndex;
        if (value < 0)            // target before middle element
            highIndex = middleIndex – 1;
        else
            lowIndex = middleIndex + 1; // target after middle element
    }
    return -1;  // not there
}
```

Search for 88 in this array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-20	4	12	22	41	42	45	48	78	90	111	403	414	508	902

What value could be returned in the last line to indicate "not there" if we wanted this method to do the same thing as the Arrays.binarySearch?

Complexity

What makes a method have O(log n) complexity? Dividing by 2 at each step until a base case < 1 is reached (the halving function in the book) leads to $\log_2 n$ steps. Dividing by 3 at each step leads to $\log_3 n$ steps. Logarithmic algorithms divide the work by a certain amount at each step. They are very fast algorithms, and often appear to run in constant time.

Contrast logarithmic algorithms with exponential algorithms, e.g., $O(2^n)$. Exponential algorithms multiply the amount of work at each step. These algorithms grow so fast that relatively small data sizes cannot be handled by even the fastest computers.

Quadratic Sorting Algorithms (Section 12.1)

Selection Sort

Selection sort works by repeatedly finding the smallest item in the unsorted portion of the array and swapping it into its correct position. The unsorted portion then decreases by one element. The sorted portion builds up from `data[0]` through `data[n-1]`, one element at a time. Your instructor will show you how selection sort works on the following data array.

100	25	-12	82	16	179	-85	-212	75	14

Note that the author of our book chooses to find the largest element in the unsorted portion of the array at each step and swap it into its correct position at the right-hand end of the array.

Complexity analysis of selection sort:

At the first step, how many comparisons are done to find the smallest item? __________

At the second step, how many comparisons are done to find the smallest item? _________

In the last step, how many comparisons are done to find the smallest item?__________

What is the solution to this equation?

$$\sum_{i=1}^{n-1} i$$

What is the worst-case and best-case complexity of selection sort? ________________

Write a method called `minimumPosition` which receives an array of `ints` and an index from which to begin the search, and which returns the index at which it finds the minimum value in that portion of the array.

```
private static int minimumPosition(int[] array, int from)
{















}
```

Insertion Sort

Insertion sort maintains a sorted portion of the array at the beginning of the array, just as selection sort does. This sorted portion grows by one element at each step, but the way it grows is different from the way it grows in selection sort.

At each step, insertion sort examines the first element in the unsorted portion of the array (call it *current)*, and shifts elements from the sorted portion to the right in order to insert *current* into its proper place in the sorted portion.

Here are elements in an array. The vertical line marks the divide between the sorted portion and the unsorted portion. At the very beginning, the sorted portion contains only one element.

15 | 12 3 7 10 14 17 11 2

Now it is time to put 12 into its proper position. Imagine pulling 12 out, then sliding 15 over one to the right and putting 12 back in at the beginning. Now the array looks like this, and the sorted portion has grown by one element:

12 15 | 3 7 10 14 17 11 2

At the next step, 3 comes out then 15 slides over one, 12 slides over one, and 3 goes back in at the beginning. Finish running the algorithm on this array.

12	15	3	7	10	14	17	11	2

Complexity analysis of insertion sort

Let's think worst case (which happens when the array is already sorted, but it's sorted in reverse.)

How many comparisons happen in the first step? ______

In the second step? _______

How many comparisons happen in the very last step, when you have to put the element at data[n-1] into its proper position? _______

What is the worst case complexity of insertion sort? _________________________

Unlike selection sort, which always does the very same thing regardless of how the data comes in, insertion sort performs very well if the data happens to already be sorted. In this case, insertion sort never does any shifting at all. How many comparisons of one element to another would insertion sort have to do if the data comes to it already sorted?

____

It is sometimes the case that an application keeps a data set sorted, but periodically tacks on new data at the end of the array. If this is how data is managed, insertion sort is an excellent choice for a sorting algorithm to re-establish the sortedness of the data. Why?

Bubble Sort

Trace this loop on the data below.

```
for (int j = 0; j < data.length - 1; j++)
{
    if (data[j+1] < data[j])
    {
        temp = data[j];
        data[j] = data[j+1];
        data[j+1] = temp;
    }
}
```

22 70 13 91 -5 43 51 85 2 15 49

What does the code do to the array of data?

Now suppose that we run this loop again and again. Can we eventually sort the array using this basic idea?

How might we save some steps?

Now let's count comparisons, assuming that we stop at the end of the unsorted portion at each pass.

How many comparisons are done in the first step? ________

In the second step? _______

So we again have a sum from 1 to n-1, giving bubble sort a complexity of ____________.

Rewrite the above loop so that it starts at the end of the array and bubbles the smallest element to the beginning. That's what you will have to do in our lab this week.

Mergesort (Section 12.2)

Steps of the Algorithm:

1. Divide the list in half evenly.
2. Base case: If the portion of the list to sort is less than two items, just return.
3. Call Mergesort recursively to sort the first half of the list.
4. Call Mergesort recursively to sort the second half of the list.
5. Merge the two sorted lists.

Notes:

1. Arguments to the Mergesort method: the list to be sorted, the left index of the portion of the array to be sorted, and the number of elements to sort. Mergesort would initially be called with the data array, index 0, and data.length, assuming we wish to sort the entire list.
2. If the list doesn't divide evenly, the second recursive call gets the one extra (it really doesn't matter which one does).
3. Since we are using recursion, we need a base case for falling out of the recursion. The base case happens when the list has fewer than two elements.
4. One smart thing, which your book does not do, is to do a quick check after the recursion to see if the largest item in the first half of the list is less than or equal to the smallest in the second list. If so, then the list is already sorted, then no merge is needed. This seems unlikely, but if the data had already been sorted, then the complexity of the sort is improved.
5. The book's code calls a non-recursive merge method that merges the two halves using a temporary array. Then it copies back to the original array. This seems inefficient, but it is the standard way that Merge Sort is done. The copying does not increase the big-Oh of the merge portion of the code.

Merge:

The parameters to the merge method are the data array, the index where the first half begins, the number of elements in the first half, and the number of elements in the second half.

Imagine that we have recursively sorted two halves of the following array and we want to merge the two parts into a temporary (sorted) array and then copy the contents of the temporary array back to the original. We keep three indexes. Let's do the work.

15	32	34	45	56	72	12	35	39	55	81	89	95

temporary array:

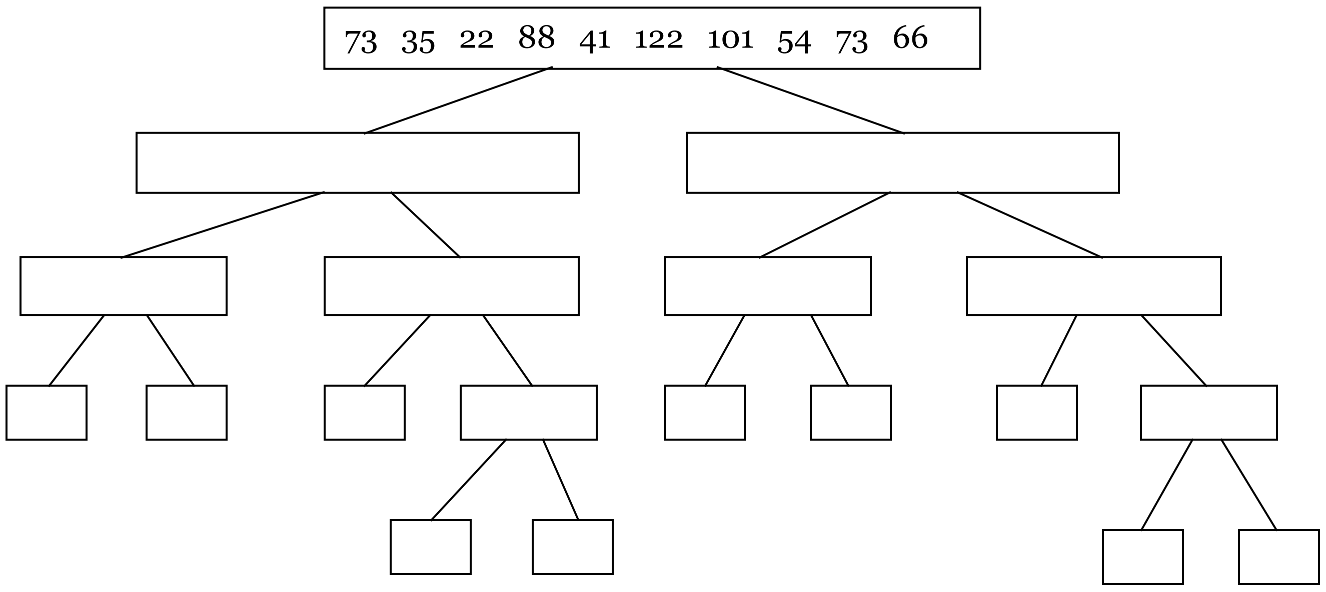
Notes:

1. As long as both lists still have elements to merge, the tops of the two sorted sublists are compared, and the smaller element is placed into the temporary array. Three indices are needed to keep track of where we are in each of the arrays.
2. We come out of the while loop when one of the lists is empty.
3. If the first list still has elements, copy the rest of the first list to the temporary array.
4. If the second list still has elements, there is no need to copy them to the temporary array, because they are already in the correct spots in the data array.
5. Copy the temporary array back to the original. You should only copy back the elements that were put into the temporary array during this recursive call.
6. Many times an overloaded non-recursive Mergesort with fewer parameters is set up to initialize the algorithm. A normal argument for a sort is the array to be sorted. Here is a non-recursive Mergesort that has only the data array as a parameter. It calls the recursive one with the appropriate values to sort the entire array.

```
public static <T extends Comparable<?super T>>  void mergeSort( T [] data)
{
    mergeSort(data, 0, data.length);
}
```

Using Recursion Tree to Illustrate Merge Sort

Page 638 of your book shows a general recursion tree (with empty rectangles to show how the data array is divided up. Draw the tree of recursive calls performed by Mergesort during the process of sorting the following data array. Show the recursive calls performed by the Mergesort during the process of sorting the following array:



Complexity of Mergesort

Recursive algorithms often are called divide-and-conquer algorithms because they divide the work into smaller problems of the same type, solve the problems on the smaller data (with recursion), and finish conquering the problem by combining the results from the sub-problems.

Book's Informal Analysis: The book argues that we do O(n) work at each level of the recursion tree (almost all elements are involved in a merge at each level of the tree). So the complexity is the work at each level multiplied by the number of levels of the tree. How many levels are there in the recursion tree for an array with n elements? _______.
The number of levels in the recursion tree is the same best case and worst case. The complexity for Mergesort is ____________.

The informal analysis is a good way of thinking about the complexity of Mergesort. Here is a more advanced analysis using recurrence equations. It is okay if you do not understand perfectly. See how much you can get.

First let's review functions. Consider the function $T(n) = n^2 + \text{sqrt}(n)$.

- What is T(4)? _______________
- What is T(x)? _______________
- What is $T(\frac{n}{2})$? _______________

What about a recursive function like the following?

$T(1) = 1$

$T(n) = 3 * T(\frac{n}{2}) + n^2$, for $n > 1$.

- What is T(2) __________________
- What is T(x) for $x > 2$, ______________
- What is $T(\frac{n}{2})$ for $\frac{n}{2} > 2$, ___________________

Our first step in analyzing Mergesort is to write a function T(n) describing the time it takes Mergesort to sort n items.

Consider the four broad original tasks:

1. Divide the list in half.
2. Recursively sort n/2 items.
3. Recursively sort another n/2 items.
4. Merge the items.

Task 1 (Dividing): Dividing the list in half is constant with respect to the size of the list. It takes the same amount of time for a computer to find the middle of the list, no matter what the size of the list is. (Note that it would not be constant for a linked list.) So this step takes O(1) time.

Task 4 (Merging and Copying): Consider one step in the merge to be "look at the top of the two lists, and put the smaller item onto the new list." This one step takes constant time. How many times do we do this step on a list of size n? Sometimes we can save a few comparisons, but worst case we have O(n) steps to copy elements into the temp array. We also have O(n) steps to copy the temp array back to the original.

Tasks 2 and 3: How long does the recursion take? We will describe the recursion time using our function T. The time it takes to sort a list of size n is $T(n)$. The time it takes to sort half the list is $T(\frac{n}{2})$.

$T(n) = O(1) + O(n) + 2T(\frac{n}{2})$

Clearly O(n) is greater than O(1), so we can disregard O(1). But what about $2T(\frac{n}{2})$? How do we turn this into an order of complexity?

We need to solve this recurrence relation.
STEP **1**:
$T(n) = 2T(\frac{n}{2}) + n$

STEP **2**:
What is $T(\frac{n}{2})$? Go back to the original function description and plug in $\frac{n}{2}$ for n.
$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$.

Substitute: this into the original equation

$$\begin{aligned} T(n) &= 2\,T(\tfrac{n}{2}) + n \\ &= 2\,(2\,T(\tfrac{n}{4}) + (\tfrac{n}{2})) + n \\ &= 4T(\tfrac{n}{4}) + n + n \\ &= 4T(\tfrac{n}{4}) + 2n \end{aligned}$$

STEP **3**: What is $T(\frac{n}{4})$? $T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$. Substitute again.

$$\begin{aligned} T(n) &= 4T(\tfrac{n}{4}) + 2n \\ &= 4\,(2\,T(\tfrac{n}{8}) + (\tfrac{n}{4})\,) + 2n \\ &= 8T(\tfrac{n}{8}) + n + 2n \\ &= 8T(\tfrac{n}{8}) + 3n \end{aligned}$$

Do you see a pattern? On the first step, there was a 2 being multiplied times the T value and there was a 2 in the denominator of the T parameter. On the second step, those values were 4's. On the third step, those values were 8's. They are all powers of 2. And whatever power of 2 we have in those two places, we are multiplying n by the power. What will happen at the *kth* step? Fill in the simplified form at step k (think of k as the power).

STEP k:
T(n) = _____T ($\frac{n}{\_\_\_}$) + ____ n

To solve the recurrence, we want to get rid of the recursion in the formula. We need a base case! Ah, we have one. We know T(1), the time Mergesort takes on a list of size 1. That is constant time. We will recurse until we hit the base case where T($\frac{n}{2^k}$) will be T(1). That happens when the numerator n equals the denominator 2^k.

When does n = 2^k? When k = $\log_2 n$. At step $\log_2 n$, we can get out of the recursion.

STEP $\log_2 n$:
T(n) = n T(1) + ($\log_2 n$)(n) = n × constant + n × $\log_2 n$
Which is larger? n × constant or n × $\log_2 n$? The latter! So the algorithm is O(nlogn).

To set up a recurrence equation for the time of a divide-and-conquer recursive algorithm:

T(n) = cost of dividing into subproblems +
cost of recursion +
cost of combining the recursive solutions

Since the code for the dividing and the code for the combining are non-recursive, we can see which is greater, and select it.

T(n) ≤ constant * max(cost of dividing, cost of combining) + cost of the recursion

<u>If the recursive calls are all on approximately the same size data:</u>
T(n) = max(cost of dividing, cost of combining) + a T(n/b)
Here, a and b are variables: The variable *a* refers to the number of recursive calls, the n/b refers to the size of the data worked on by the recursive calls. Note that Mergesort's combining was the costliest of dividing vs. combining. Next time we'll see Quicksort. It has a constant combine, but a more costly divide.

Quicksort (Section 12.2)

The General Algorithm:

1. Select an element that might be the median element. Call it pivot
2. Change the array around so that elements less than (or <=) the pivot are put on the left side of the array, and elements greater than (or >=) the pivot are put on the right side of the array.
3. Put the pivot element where it belongs (everything to the left of the pivot is <= pivot and everything to the right of the pivot is >= the pivot).
4. Recursively sort the part of the array that is left of the pivot location.
5. Recursively sort the part of the array that is right of the pivot location.

Notes:

1. The parameters to Quicksort are the array, the *first* index (where sorting should begin), and the size *n* (the number of elements to sort).
2. Since this method is recursive, there must be a base case. The book's base case occurs if there are fewer than two elements. If there are fewer than two elements, Quicksort returns. (Let me suggest another base case: if there are two elements, the elements are swapped if they are out of order, and then Quicksort returns.)
3. Otherwise, the partitioning begins (partitioning is a term used for dividing the array into parts: the elements less than (or equal to) the pivot first, then the pivot, and finally the elements greater (or equal to) the pivot. Although it seems counter-intuitive, allowing duplicates to go either to the left or the right is best if there are lots of duplicates.
4. Steps 1, 2, 3, and 4 are the partitioning part, and the author of the code puts them into a separate method called *partition*. The *partition* method does almost all of the work of the algorithm. It picks a pivot, puts smaller elements (and perhaps some duplicates) to the left, the larger (and perhaps duplicates) to the right, and then puts the pivot in place. It also returns the index where the pivot ends up. We will discuss that method below.
5. After *partition* does its work, Quicksort recursively sorts the part of the array that is to the left of the pivot, and recursively sorts the part of the array that is to the right of the pivot. Once the recursion is done, Quicksort returns.
6. Some implementations of Quicksort have a much broader base case. They will call another sort on the data when the size of the subarray is less than some value, say 10 or 15. They will run an Insertion Sort or other simple sort on the small sub-lists. Let's use two or less as the base case for now.

P**artition** (a slightly different implementation than the book describes)

1. The first task of the partition method is to choose a pivot. The goal is to choose an element that would be the real middle when this portion of the array is sorted. Our textbook uses the first element. This method will produce a very poor pivot if the array is already sorted. Instead, we will use the medianOf3 algorithm: sort the first, middle, and last element in the portion of the array, and choose the middle element as the pivot. Even though the pivot may be poor occasionally, it is unlikely that the pivot will continue to be poor each time. Other implementations of Quicksort may use different algorithms for choosing the pivot.

2. We will swap the middle element (pivot) with the second element from the end (this is sometimes called *hiding the pivot,* but we'll remember where it is). From now on, we will basically use the pseudocode from the text.
3. Instead of using a temporary array, Quicksort implementations use a clever method to put the smaller elements before the larger ones. They scan from the left of the array (actually the left of the portion to be sorted) until they find something greater than or equal to the pivot. The code uses the index variable *tooBigIndex*. Then they scan down from the right, using variable *tooSmallIndex,* until they find something less than or equal to the pivot. Then the values at *tooBigIndex* and *tooSmallIndex* are swapped.
4. This scanning procedure then continues with *tooBigIndex* moving up from its current position and *tooSmallIndex* moving down. Finally when *tooBigIndex* and *tooSmallIndex* cross, we're done.
5. The pivot is then put in place. Its place is where the *tooBigIndex* variable is, so to put the pivot in place, the *last-1* element is swapped with the *tooBigIndex* element.

Try out the partition method described above, called with `partition(data, 0, 15)`, on the three arrays shown below. Show how the data ends up at the end of the partition method. Also, specify the return value from partition. The code for partition is shown on the next page.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
100	38	50	77	19	27	2	48	46	95	104	-8	15	55	30

Pivot = _____ Return Value = _____

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
40	9	71	52	14	22	60	68	58	100	6	90	1	-10	60

Pivot = _____ Return Value = _____

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
93	2	-14	99	119	89	102	122	-12	95	90	741	-6	80	-10

Pivot = _____ Return Value = _____

```
public static int partition(int [] data, int first, int n)
{
    // Step 1:  sorts the first, middle, and last elements
    int mid = first + n/2;
    int last = first + n - 1;
    if (data[mid] < data[first])
        swap (data, first, mid);   // swaps data[first] with data[mid]
    if (data[last] < data[mid])
        swap (data, mid, last);
    if (data[mid] < data[first])
        swap (data, first, mid);

    int pivot = data[mid];
    // hide the pivot in the second to the last position
    swap(data, mid, last-1);

    int tooBigIndex = first+1;
    int tooSmallIndex = last-2;

    for (;;)
    {
        while (data[tooBigIndex] < pivot)
            tooBigIndex++;
        while (data[tooSmallIndex] > pivot)
            tooSmallIndex--;
        if (tooSmallIndex <= tooBigIndex)
            break;
        swap (data, tooBigIndex, tooSmallIndex);
        tooBigIndex++;
        tooSmallIndex--;
    }

    // put the pivot in its place in the array
    data[last-1] = data[tooBigIndex];
    data[tooBigIndex] = pivot;
    // return the index where the pivot ends up
    return tooBigIndex;
}

public static void quickSort(int[] data, int first, int n)
{
    int pivotIndex;
    int n1, n2;

    if (n <= 1)
        return;
    if (n == 2)
    {
        if (data[first+1] < data[first]) {
            swap (data, first, first+1);
        }
        return;
    }
    pivotIndex = partition(data, first, n);
    n1 = pivotIndex - first;
    n2 = n - n1 - 1;
    quickSort(data,first,n1);
    quickSort(data,pivotIndex+1, n2);
}
```

Complexity of Quicksort

1. Choosing the pivot is O(_____).
2. Handling the base case(s) is O(______).
3. The code for *partition* has nested loops, so it looks perhaps $O(n^2)$, but we don't look at each of the elements every time through the outer loop. We basically look at each element once. The method *partition* is O(______).
4. What about the recursion? We called the recursion twice, each time with a smaller array. Suppose we were fortunate enough to split the list in half every time, so that the recursion was called once on the first half, and then on the second half.

T(n) = cost of stuff before calling partition + cost of partition + recursivePart
$T(n) = O(1) + O(n) + 2T(\frac{n}{2})$

Where have we seen this before? It is the same recurrence as Mergesort. This time we did O(n) work before the two recursive calls, and constant afterward. With Mergesort one does constant work before the recursive calls, and O(n) afterwards.

The complexity of Quicksort depends on a good split of the data. IF we did a bad split, instead of splitting the list into two equal parts, we could split the list into one list of size n-1. The other list would be easy to sort (a base case), but then we get this recurrence.

T(n) = O(1) + O(n) + complexityofSortingBaseCase + complexityofSortingRest
$T(n) = O(1) + O(n) + O(1) + T(n-1)$

Let's solve: $T(n) = \underline{T(n-1)} + n$ --- What is T(n-1) ? $T(n-1) = T(n-2) + n-1$.
$= T(n-2) + n-1 + n$
$= T(n-3) + n-2 + n-1 + n$
$= T(n-4) + n-3 + n-2 + n-1 + n$
...
$= T(n-(n-2)) + 3 + 4 + 5 + ... + n-3 + n-2 + n-1 + n$
Remember we know T(2). It is a base case, and constant.
= approximately the sum of the numbers from 1 to n: $O(n^2)$

With some statistical analysis that we will not do, it can be shown that on average Quicksort does constant × O(nlogn) work. It is quite a challenge to find data that makes the Quicksort behave badly, when the medianOf3 pivot choice is made. Even if the pivot is poor the first time, to get the $O(n^2)$ behavior, bad pivots need to be chosen again and again. If the first element is chosen as the pivot, it is easy to find data to make Quicksort behave badly.

With the medianOf3 choice of a pivot, sorted order and reversed order arrays are best case (still O(nlogn)). The middle element is the median every time. Even though Quicksort has the potential to be an $O(n^2)$ sort, and Mergesort is guaranteed to be an O(nlogn) sort, in actual practice Quicksort performs the best. In the class account on the student machine, there is a program called Sorts.java that has the code for Selection Sort, Insertion Sort, Mergesort, and Quicksort, along with timing. Check it out:

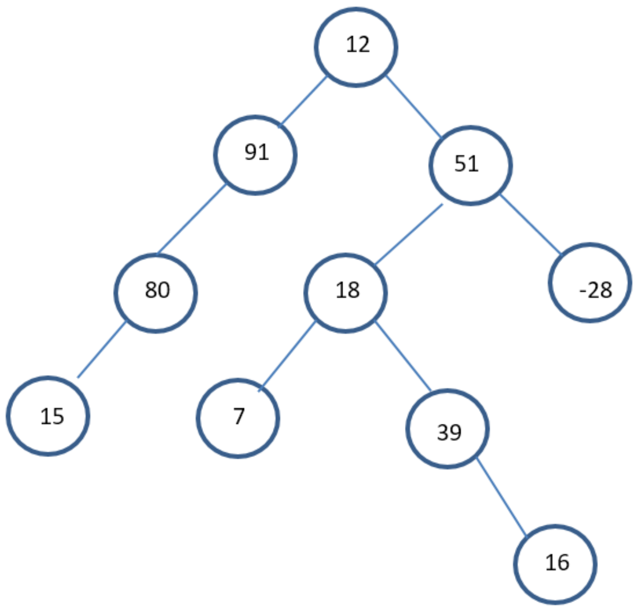
/u/css/classes/2440/Spr2015/Sorting/Sorts.java

As we have discussed before, sort methods often have the array to be sorted as the only parameter. But the recursive implementation Quicksort needs extra parameters. The implementation of a sort should not have to be known by the calling program. Here is a non-recursive method that simply sets up the recursive one.

```
static void quickSort (Object arr[])
{
    quickSort(arr, 0, arr.length); // calls the recursive one
}
```

Binary Trees (Sections 9.1, 9.2)

After reading section 9.1, and from CS 1100, you should be able to answer the following questions about this tree.



1. Is the tree a binary tree? Why or why not? ______________________________
2. What is the root of the tree? __________
3. What are the leaves of the tree? ____________________________
4. What is the parent of 80? __________
5. What is the only node with no parent? __________
6. What is the sibling of 39? ____________
7. What are the ancestors of 80? __________________
8. What are the descendants of 18? _____________________________
9. What is the left child of 51? __________
10. What is the right child of 39? __________
11. What is the depth of node 51? ___________
12. What is the depth of the root node? ____________
13. What is the depth of the tree? ________ The height of the tree? ________
14. Draw the left subtree of 51.

According to the textbook, what is the depth (or height) of an empty tree (a tree with no nodes)? ____________ **Note:** Many textbooks define depth (or height) differently from our textbook. Their depth or height will be one more than ours (ours counts the number of edges on the path from the root to the most remote leaf. Others count the number of vertices, including the root and leaf). By definition, those books will define the depth of an empty tree as 0. Does anyone remember how the discrete math book defines height (or depth) of a tree? Be careful on questions on trees on standardized tests. They should define height with their questions.

In discrete math you studied **binary search trees**. Binary search trees are very useful. They are covered in the latter part of chapter 9 in our textbook. We won't spend time on them this semester but you will study them in in CS 3460. The binary tree on the previous page is NOT a binary search tree. Why not?

Full Binary Trees

1. What is a full binary tree?
2. Is the binary tree on the previous page a full binary tree? _________
3. Draw a skeleton for a full binary tree with 15 nodes.
4. What is the height of a full binary tree with 1 node? ______ 3 nodes? _______
   7 nodes? ________ 15 nodes? ________ n nodes? ________________________
5. How many nodes will a non-empty full binary tree of height h have? ____________

Complete Binary Trees

1. What is a complete binary tree?
2. Is the tree on the previous page a complete binary tree? __________

3. Draw a complete binary tree with 12 nodes. To review some discrete math material, put some values in the tree that make it a binary search tree, too.

Storing Complete Binary Trees in Arrays

In section 9.2 of the textbook, we see that a complete binary tree can be stored in an array. We store the elements level by level, left to right, beginning in cell 0 of the array. Show how the complete binary tree that you drew on the previous page can be stored in this array:

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

Using the Indices of an Array Implementation of a Complete Binary Tree

From the array we can get the same parent/child information as we can from looking at the complete binary tree.

1. Where in the array is the parent of the node at index 6? __________
2. Where in the array is the parent of the node at index 9? __________
3. Where, in general, is the parent of the node at index k? __________
4. Where in the array are the children of the node at index 4? ________________
5. In general, where are the children of the node at index k? ____________________

If a binary tree is not a complete binary tree, it is not usually stored in an array. In section 9.2, the author shows how a binary tree can be represented with nodes that contain data and two links to left and right children. You will use a similar representation when you study binary search trees in CS 3460, but we will not use it here.

Test your understanding of trees by answering the following true/false questions.

1. (True, False) All binary trees are complete binary trees.
2. (True, False) All full binary trees are complete binary trees.
3. (True, False) All complete binary trees are full binary trees.
4. (True, False) A binary tree of height 10 might have exactly 50 nodes.
5. (True, False) A binary tree of height 50 might have exactly 25 nodes.
6. (True, False) A binary tree of height 100 might have exactly 99 nodes.
7. (True, False) A full binary tree of height 10 will have exactly 1024 nodes.
8. (True, False) A complete binary tree with 64 nodes will have height 8.
9. (True, False) A complete binary tree with 10,000 nodes will have height 13.

Binary Heap (Section 10.1)

A binary heap is NOT a binary search tree. Make sure you do not confuse the two data structures. Both use a binary tree, but they are quite different.

A binary heap is a complete binary tree with values in a certain arrangement. Using circles for nodes, draw the skeleton of a complete binary tree with 19 nodes. Do not put values in your nodes yet. We will turn the complete binary tree into a binary heap by adding those values.

Notice that at each level of the tree, other than the bottom level, there are a power of 2 nodes. Each level, other than the bottom level, has twice as many nodes as the previous level has. A complete binary tree always has as many leaves as it has other nodes. In terms of n, give an exact formula for the height of a complete binary tree with n nodes.

____

Now we will add values to your tree to turn it into a binary heap. In heap order, a parent has a value at least as large as either of its children. It is fine if the left child's value is higher than the right child's value, or vice versa. Fill in the values of the above tree so that the tree is in heap order

Priority Queues

Priority queues are data structures designed to be able to access the item of highest priority quickly. Binary heaps are used to implement priority queues more efficiently than sorted arrays, sorted linked lists, or binary search trees. Binary heaps store the highest priority item in the root where it can be accessed immediately. As items are inserted and removed from a binary heap, heap order has to be re-established. We will discuss how to do this shortly.

In the heap you made on the previous page, high values meant high priority. Sometimes higher priority is represented by lower numbers. Our textbook represents higher priority with higher numbers so that's what we will do.

Inserting Into Binary Heaps

Items to be added to the heap are inserted at the bottom in the next available slot on the bottom level of the complete binary tree. If the inserted item messes up the heap order, we keep swapping it with its parent until the resulting tree is again in heap order. At most O(log n) swaps are necessary.

Insert the following items one at a time into a binary heap:

56, 92, 14, 80, 9, 22, 105, 95, 80, 50, 70, 42

Deleting From a Binary Heap

When we remove the highest priority item from the root of the binary heap, we must fill in the hole that is left. We take the rightmost item from the bottom level of the tree and put it into the hole. Then that value is percolated down until it reaches an appropriate spot in the tree. To percolate down, we swap the item with the larger (higher priority) of its children until it either reaches the bottom level of the tree or has as high of a priority as its children. Our textbook calls this process "reheapification downward," but it is called percolating down in many books. The method that carries out the action is called deleteMax( ). Show the result of two deleteMax( ) operations on the heap you created on the previous page. Draw the heap again before doing the deletions.

Complexity Questions

A binary heap is a complete binary tree. It is usually stored in an array, as we studied in section 9.2. Recall that the goal of a priority queue is to be able to access the highest priority item quickly. Where in the array is the highest priority item of a binary heap located? _______________ What is the complexity of a peek( ) method that returns (without removing) the highest priority item in a heap? _________________

With a priority queue, data can keep trickling in. Each time another piece of data comes in, we insert it using the technique we learned earlier. Suppose we are inserting an element into a binary heap. What values (in comparison to other values in the heap) will cause us to do the most work (make the largest number of swaps)?

What is the worst-case complexity of a single insertion into a binary heap? ____________

Suppose we are building a new binary heap and several pieces of data need to be inserted. What arrangement of this data will cause more swapping, smallest to largest order or largest to smallest order?

The worst-case complexity of the total of all the insertions is what? ________________

What are the complexities of each of the following steps involved in deleting the highest priority item in a binary heap?

1. Save the highest priority item in a temporary variable. ____________________
2. Place the last item in the heap at the top. ____________________________
3. Percolate that last item down until it finds its proper place in the heap.__________

Therefore, the worst-case complexity of a single deleteMax( ) operation is ___________.

The average case is usually the same as the worst case. Why?

The following chart can help answer questions about why a binary heap is worth the trouble of implementing it, compared to using a sorted array or a sorted linked list as a priority queue. Give the big-oh complexity of each of the following methods for each structure.

	Access high priority item	Insert an arbitrary item.	Remove the high priority item.
Binary Heap			
Sorted Array			
Sorted Linked List			

There is a PriorityQueue class in the Java library. It makes smaller values have the higher priority, though it gives you options to change that. It has strange names for adding an item and removing the highest priority item. The add method is called offer( ) and the method that removes the highest priority element is called poll( ).

HeapSort (Section 12.3)

As a review of the previous material, place the following elements into a binary heap and draw the complete binary tree:

83, 14, 90, 3, 200, 17, 45, 20, 8, 50, 12, 52

Now show the underlying array.

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

Show the result of a deleteMax( ) operation.

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

Show the result of another deleteMax( ) operation.

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

Note that we don't need the last two array spots any more.

There is a cool sorting algorithm called heapSort that makes use of a binary heap. It requires that all the data be available before the sort, and it uses the idea that every time you remove the highest priority element, you end up with one more unused position in the array.

Here is the idea of the sort: First make a binary heap out of an unsorted array. Then save the highest priority element in a temporary variable and call deleteMax( ) to remove it from the heap. Now that the heap has one fewer element, the last cell of the array is unused. Store the value from the temporary variable in that location and consider that cell not to be part of the heap any more. Then repeat, shrinking the size of the heap at every step and storing the next largest element in the newest unused cell of the array. Here is the code for heapsort. Code for heapInsert( ) and deleteMax( ) is not shown.

```
public void heapSort(int [] array)
{
    // make a binary heap out of the array
    for (int i = 1; i < array.length; i++)
    {
        // assume the array from 0..i-1 is in heap order
        // insert the item at position i into the heap
        heapInsert(array, i);
    }
    // deleteMax one by one, putting the maxes in the newly unused
    // entry of the heap
    for (int i = array.length-1; i  > 0; i--)
    {
        int temp = array[0];    // temp is the current maximum item
        deleteMax(array, i);    // delete the max from a binary heap
                                // stored in array [0..i]
        array[i] = temp;        // put the removed element at the old
                                // end of the heap
    }
}
```

What is the complexity of the first for loop in the code shown above? ___________________

What is the complexity of the second for loop? ____________________

What is the total complexity of heapsort? _____________________

Final Exam Study Guide

1. Chapter 1:
   a. Pre-conditions and Post-conditions
   b. Big-oh Analysis: Be able to tell the Big-oh of a method given to you. Be able to use the Big-oh formula given in class.
   c. Boundary conditions and testing
   d. Two-dimensional arrays

2. Chapter 2: Java classes
   a. The ADT – abstract data type
   b. References vs. primitives
   c. Exceptions
   d. Clone and equals methods

3. Chapter 3: Collection Classes
   a. Implementing a Bag or a Sequence with an Array
   b. Understanding the difference between the size and the capacity with the array Implementation
   c. Understanding the big oh of the bag class operations and the sequence operations.

4. Chapter 13: Inheritance:
   a. Be able to extend a class. Keywords: super, protected, instanceof
   b. Know what it means to override a method
   c. Narrowing and widening conversions – what assignments are allowed
   d. Be able to write a class that extends another
   e. Abstract classes and Interfaces. Be able to write an abstract class or an interface. Be able to write a class that implements an Interface.

5. Chapters 4 : Linked Lists
   a. Be able to trace code with the Node class.
   b. <u>Be able to write some linked list code</u>. Example: the question on the last test that asked you to write the Stack class.
   c. Know what operations are more efficient on a linked list and what operations are more efficient on an array.

6. Chapter 5: Generics
   a. Know what is involved in changing a non-generic class or method to a generic one.
   b. Be able to write a small generic class.
   c. Be able to use a generic node class.
   d. Know the methods in the interfaces: Iterable<T>, Iterator<T>, Comparable<T>. Know when these interfaces are needed.

7. Chapter 6 : Stacks
    a. Know what a stack is, and what the common stack operations are. Know what LIFO stands for.
    b. Understand the Stack applications we studied: balanced parenthesis, infix to postfix.
    c. Know the big-oh of the standard stack operations.

8. Chapter 7: Queues
    a. Know what a queue is, and what common queue operations are.
    b. Understand how a queue was implemented as an array (and how the data could end up wrapping around from the back of the array to the front).
    c. Understand the linked list implementation of a queue, and why it is important to have a reference to the tail of the queue.
    d. Know the big oh of the standard queue methods.

9. Chapter 8: Recursion
    a. Be able to trace recursive code and tell what it does.
    b. Write a recursive method to do something. There is a recursion section on Javabat if you want to practice.

10. Chapter 11: Searching
    a. Understand the serial and binary searches and the code that goes with them.
    b. Know the complexity of the binary search, and understand what makes a method have a complexity of log(n). Make sure that you understand that the underlying array must be sorted in order for a binary search to be done.
    c. Hash tables – understand the goal of hashing. Be able to place some elements in a hash table, given a hash method. Be able to use linear probing, double hashing, or chaining to handle collisions.
    d. Understand what the hashCode() method is supposed to do : the method that is used by the Java library HashMap, HashSet, and HashTable classes.

11. Chapter 12: Sorting
    a. Selection Sort – be able to show how it works. Know its complexity.
    b. Insertion Sort – be able to show how it works. Know its complexity.
    c. Bubble Sort – be able to show how it works. Know its complexity.
    d. Merge Sort – be able to show how it works. Know its complexity.
    e. Quick Sort – be able to show how it works. Know its complexity. Be able to trace an array using the partition method given in class on Wednesday.

Homework: Two-dimensional Arrays

Name ____________________________

1. Write a void method, *showMap,* that takes a 2-dimensional array of boolean as a parameter and prints the array in the following tabular format: if the corresponding entry is true, then an 'X' is printed, otherwise a space (' ') is printed.

2. Write a method *createMines* that creates and returns a 10x10 array of boolean. Set 10 random entries in the table to true. The rest of the entries should be false.

3. Write a method, *findAverages,* which takes a 2-dimensional array of double as a parameter and returns a 1-dimensional array of double containing the averages of each row of the2-dimensional array.

4. Write a boolean method, *noDuplicate,* which takes a 2-dimensional array of int as a parameter and returns true if there are no two entries in the array that are the same and false otherwise.

Homework: Big-oh Analysis

Name ______________________________

1. Joe Smoe wrote a method to evaluate his Steiner Tree class. The method is *$O(n^2)$* where *n* is the number of vertices in the tree. When *$n = 1{,}000$* his program takes *35* time units. How long would you expect the program to take when *$n = 5{,}000$*? _______________

2. Graduate student Linda Hand is also working on the Steiner Tree problem. She claims the following method *isConstant* is constant because it has no loops. In fact it only has one line - a call to the *contains* method on neighborList which is an ArrayList object. What is wrong with Linda's claim?

```
public boolean isConstant (Neighbor j) {
    return neighborList.contains(j);
}
```

3. Geraldine's assignment is to write either a SelectionSort or a QuickSort. She selects one of the two sorts and implements it correctly. When she runs the sort on a list of length n = 5,000, the sort takes 20 time units. When she runs the sort on a list of n = 100,000, the sort takes 525 time units. SelectionSort is $O(n^2)$ and QuickSort is O(nlogn). Which sort do you think she implemented? Why?

4. Ryan's puzzle program is $O(2^n)$. It still runs fairly fast, a half minute or so, when n = 20, but he needs to run it on n=100. He has tried running the program for over two weeks and it still hasn't come back. "I know what I'll do," says Ryan. "I'll use a network of 5 computers, and give each computer one fifth the work." Even assuming that Ryan can divide the work evenly between the five computers, what is wrong with his reasoning?

5. What is the running time of the following methods?

```
public static void first (double arr[])
{
    int count = arr.length;
    int middle = count / 2;
    if (count > 0)
    {
        for (int i = 0; i < middle; i++)
            if (arr[i] < arr[middle])
                arr[i] = Math.random();
    }
}
```

Running Time: ______________________________

```
public static int second(int n)
{
    int number=0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            number++;
        }
        for (int k = 1; k < i; k++)
        {
            number = number + i * i - k;
        }
    }
    return number;
}
```

Running Time: ______________________________

```
public static void third(int arr[]) // n is the length of the array
{
    int k, index;
    for (int i = 0; i < arr.length; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            k = 0;
            while (k < i*j)
            {
                index = (i+j+k) % arr.length;
                arr[index] += 2*i*j - k*k;
                k++;
            }
        }
    }
}
```

Running Time: ______________________________

Homework: Exceptions

Name ______________________________

All true/false are worth 5 points each. Your name above is worth 10 points. Problems on the back have point values shown.

Part I. True or False.

1. ______ If you write a method that might cause a checked exception, you must wrap the throw code in a try/catch block or declare the exception in the method's header (state in the header that the method throws an exception).

2. ______ If you write a method that might cause a runtime exception, you must wrap that code in a try/catch block or declare the exception in the method's header.

3. ______ A finally block is required after a catch block.

4. ______ Handling an exception is referred to as "ducking" the exception.

5. ______ If an exception is thrown, the remaining code in the try block is skipped.

6. ______ If an exception is not thrown in a try block, the corresponding catch block is skipped.

7. ______ If a try block has multiple catch blocks, the order of the catch blocks in the code is irrelevant.

8. ______ Runtime exceptions are exceptions that cannot be handled.

9. ______ If method A calls a method B that can throw an exception, and method A declares the exception (in the header states that it throws the exception), then method A does not have to put the call to method B in a try/catch block.

10. ______ If you write your own exception class, it must be a subclass of some other exception class.
11. ______ Both of the catch blocks below do the exact same thing (i.e., it doesn't matter if you use e or ex).

```
catch(Exception e)
{
    System.out.println (e.getMessage());
    System.exit(0);
}

catch(Exception ex)
{
    System.out.println (ex.getMessage());
    System.exit(0);
}
```

Part 2: Short answer.

1. What is the difference between throwing an exception and handling an exception? (5 pts)

2. Under what circumstances do you have to catch an exception in the code you write? (5 pts)

3. Suppose we need to write code that receives String input from a user, and we expect the String input to contain a double value (inside quotes). We want to convert the String to a double using the static method Double.parseDouble(String s) from the Double class. The parseDouble method throws a NumberFormatException if the String s is not a parsable double. Write a method printSum that takes two String parameters (that hopefully have parsable doubles in them), and either prints the sum or prints that the inputs were invalid by handling the exception in a try/catch block and responding accordingly. (20 pts)

4. Suppose you did not put the code for parsing the double into a try/catch block. Would you get a compiler error? If not, what would have happened if the method was called with bad input? (5 pts).

Homework: Bags and Sequences

Name______________________________

1. The invariants of a class are written for the programmer. The invariants are statements about the instance variables (fields) of the class. The programmer who is writing a method can depend on the invariants being true at the beginning of the method, and he/she must make sure they are true when the method finishes. What are the invariants for each of the instance variables in the IntArrayBag class:
   a. Invariant for *data*:

b. Invariant for *manyItems*:

2. The elements in the bag are stored in an int array called *data*, and *manyItems* tells how many items are stored in the bag. Why do we need the instance variable *manyItems*? In other words, why can't we just use *data.length*?

3. In the *add*, *addAll*, and *addMany* methods, what happens when *data.length* is not large enough to support the addition of the items?

4. What does the *ensureCapacity* method do when it is called with a value that is less than the length of the *data* array?

5. In chapter 2 of our text (p.83-84), the book gives instructions on how to write a *clone*() method. The *clone* method on page 139 has one additional statement that the *clone*() in the Location class did not have.

   a. What is the additional statement?

   b. In general, when you are writing a clone() method for a class, when are these additional statements needed?

6. Let *bag* be an *IntArrayBag* with fifteen items (in *data[0]..data[14]*). Suppose that the remove method is called and the first occurrence of the item to remove is at index 2. What position(s) in the *data* array will change, and how will they change?

   How will *manyItems* change?

7. In lab this week, you will implement the Sequence ADT described on pages 145-159 of your textbook. It is important that you thoroughly understand the ADT. In addition to the methods listed in your textbook, you will include the two methods toString() and equals() as described below:

```
/**
 * Returns a String representation of this sequence.  If the sequence is
 * empty, the method should return <>.  If the sequence has one item,
 * say 1.1, and that item is not the current item, the method should
 * return <1.1>.  If the sequence has more than one item, they should
 * be separated by commas with a following space, for example:
 * <1.1, 2.2, 3.3>.  If there exists a current item, then that item
 * should be surrounded by square brackets.  For example, if the second
 * item is the current item, the method should return: <1.1, [2.2], 3.3>.
 *
 * @return a String representation of this sequence.
 */
public String toString()

/**
 * Determines if this object is equal to the other object.
 *
 * @return true if this object is equal to the other object, false
 *         otherwise
 */
public boolean equals(Object other)
```

ASSIGNMENT: Look at the following program, StudentAssignment.java, which uses a DoubleArraySeq. The program creates a DoubleArraySeq and calls some of its methods. Fill in the blanks in the comments to show the values of the variables at the end of the statements (note that some of the if statements will have false conditions, and the variables inside will not change.) By the way, an empty sequence has no current element.

```
public class StudentAssignment {
    public static void main(String[] args) {
        DoubleArraySeq seq = new DoubleArraySeq();
        int answer = 1;
        int size = 0;
        double current = 0.0;
        String content = "";
        boolean isCurrent = false;
        int capacity;

        size = seq.size();                     //  1. size =  _______________

        capacity  = seq.getCapacity();         //  2. capacity =  ____________

        isCurrent = seq.isCurrent();           //  3. isCurrent =  ___________
        if (isCurrent)
           current = seq.getCurrent();         //  4. current = ______________
```

```
content = seq.toString();                  //  5. content =  _____________
seq.trimToSize();
capacity = seq.getCapacity();              //  6. capacity =  ____________
seq.ensureCapacity(5);
capacity = seq.getCapacity();              //  7. capacity =  ____________
seq.addAfter(1.1);
content = seq.toString();                  //  8. content =  _____________

size = seq.size();                         //  9. size = _________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 10. current = ______________
seq.addBefore(2.2);
content = seq.toString();                  // 11. content =  _____________

size = seq.size();                         // 12. size =  ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 13. current = ______________
seq.addAfter(3.3);
content = seq.toString();                  // 14. content = ______________

size = seq.size();                         // 15. size = _________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 16. current = ______________
seq.advance();
content = seq.toString();                  // 17. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 18. current = ______________
seq.advance();
content = seq.toString();                  // 19. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 20. current = ______________
seq.addBefore(4.4);
content = seq.toString();                  // 21. content = ________________

size = seq.size();                         // 22. size = _________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 23. current = ______________
seq.advance();
content = seq.toString();                  // 24. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 25. current = _______________
seq.advance();
content = seq.toString();                  // 26. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 27. current = ________________
seq.advance();
content = seq.toString();                  // 28. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 29. current = ________________
seq.advance();
content = seq.toString();                  // 30. content = ________________
if (seq.isCurrent())
   current = seq.getCurrent();             // 31. current = ________________
```

```
        seq.start();
        content = seq.toString();               // 32. content = ________________
        if (seq.isCurrent())
           current = seq.getCurrent();          // 33. current = ________________
        seq.advance();
        content = seq.toString();               // 34. content = ________________
        seq.removeCurrent();
        content = seq.toString();               // 35. content = ________________

        size = seq.size();                      // 36. size = ___________________
        if (seq.isCurrent())
           current = seq.getCurrent();          // 37. current = ________________
        seq.removeCurrent();
        content = seq.toString();               // 38. content = ________________
        if (seq.isCurrent())
           current = seq.getCurrent();          // 39. current = ________________
        seq.removeCurrent();
        content = seq.toString();               // 40. content = ________________
        if (seq.isCurrent())
           current = seq.getCurrent();          // 41. current = ________________

        content = seq.toString();               // 42. content = ________________
        seq.start();
        content = seq.toString();               // 43. content = ________________
        seq.removeCurrent();
        content = seq.toString();               // 44. content = ________________
    }
}
```

Homework: Inheritance

Name_______________________________________

1. CuckooClock and GrandfatherClock are both subclasses of Clock. The following statements occur in a correct program:

```
Clock myClock;                          // line 1
CuckooClock myClock2;                   // line 2

myClock = new CuckooClock();            // line 3
myClock.tellHour();                     // line 4

myClock = new GrandfatherClock();       // line 15
myClock.tellHour();                     // line 16
```

   a. What is the static type of myClock? ___________________________________

   b. What is the static type of myClock2? __________________________________

   c. What is the dynamic type of myClock when line 4 is executed? ________________

   d. In all three classes (Clock, CuckooClock, and GrandfatherClock) there is a tellHour method. In which class does the **<u>compiler</u>** look for a tellHour method that is called on the myClock? ____

   e. When line 4 is <u>executed</u>, which tellHour method will be executed: the one in Clock, the one in CuckooClock, or the one in GrandfatherClock? ________________________

   f. When line 16 is <u>executed</u>, which tellHour method will be used: the one in Clock, the one in CuckooClock, or the one in GrandfatherClock? ________________________

2. The Book class constructor has the following signature:

```
public Book(String author, int numPages)
```

   Write the <u>class wrapper</u> and <u>constructor</u> for a class called ReferenceBook which is a direct subclass of Book and which contains one String field specifying a subjectArea. The ReferenceBook constructor takes three parameters: the author, the number of pages, and the subject area. The ReferenceBook constructor's statements must cause a new ReferenceBook object to be initialized with author, pages, and area.

3. In the following pairs of classes identify the <u>superclass</u> by **circling it**, or write **NONE** if no inheritance relationship is clearly present. Write an explanation if necessary to clarify your answer.

   Cereal, Vitamin

   DVDPlayer, VCRPlayer

   Refrigerator, Appliance

   Fruit, Food

   Cup, Saucer

   Clothing, Coat

   Camera, Picture

4. If a child class does not provide any constructors, what constructors does it inherit from the superclass?

5. If a child class does provide constructors, what constructors does it inherit from the superclass?

6. Assume we have four classes: Pet, Dog, Cat and Poodle. Dog and Cat are both subclasses of Pet. Poodle is a subclass of Dog. Which of the following assignment statements are syntactically legal given the declarations shown?

```
Pet p1;
Pet p2;
Poodle poodle1;
Dog d1;
Cat c1;
```

```
d1 = new Pet();          legal      illegal
p2 = new Poodle();       legal      illegal
poodle1 = new Dog();     legal      illegal
p1 = new Cat();          legal      illegal
p1 = new Poodle();       legal      illegal
p1 = (Poodle) d1;        legal      illegal
p2 = (Poodle) c1;        legal      illegal
c1 = p1;                 legal      illegal
d1 = poodle1;            legal      illegal
poodle1 = p2;            legal      illegal
poodle1 = d1;            legal      illegal
```

Homework: Linked Lists #1

Name_______________________________________

What is the output of the following program? Draw pictures to show the linked list and its references as they change.

```
public class Animals
{
    public static void main(String args[])
    {
        String arr[] = {"pig", "cow", "dog"};
        Node p,q,r,s;

        p = new Node ("cat");
        q = p;
        r = p;

        p = new Node(arr[0]);
        p.setLink(q);

        q = new Node(arr[1]);
        q.setLink(p);

        p = new Node(arr[2]);
        p.setLink(q.getLink());
        q.setLink(p);

        if (r != null)
            System.out.println ("r: " + r.getData());
        if (p != null && p.getLink() != null)
            System.out.println
            ("p: " + p.getData() + " " + (p.getLink()).getData());

        p.getLink().setData(p.getData());

        for (s = q; s != null; s = s.getLink())
            System.out.println(s.getData());
    }
}
```

Homework: Linked Lists #2

Name______________________________________

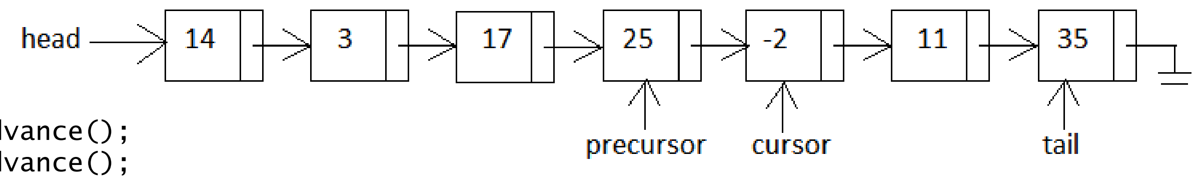
What is the output of the following program? Draw pictures to show the linked list and its references as they change.

```
public class Cities
{
     public static void main(String args[])
     {
        String arr[] = {"Paris", "Rome", "London"};
        Node p,q,r;

        r = new Node ("Madrid");
        p = r;
        q = null;
        for (int i = 0; i < 3; i++)
        {
           q = new Node (arr[i]);
           if (i % 2 == 0)
           {
               r.setLink(q);
               r = q;
           }
           else
           {
               q.setLink(p);
               p = q;
           }
        }

        System.out.println ("Answer 1: " + p.getData());
        System.out.println ("Answer 2: " + q.getData());
        System.out.println ("Answer 3: " + r.getData());

        q = p.getLink();
        p.setLink(new Node("Athens",q));

        System.out.print ("Answer 4: List: ");

        for (q = p; q != null; q = q.getLink())
        {
            System.out.print (q.getData() + " ");
        }
        System.out.println();

     }
}
```

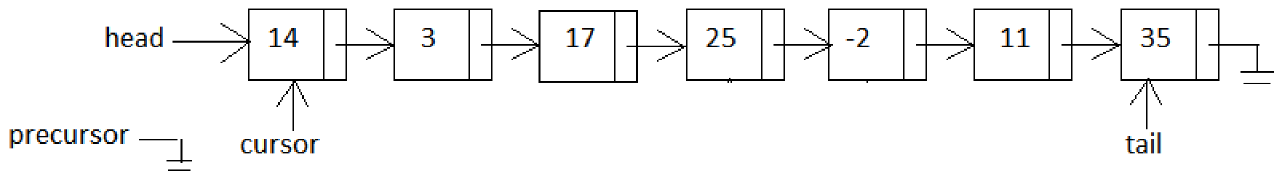
Homework: DoubleLinkedSeq

Name_______________________________

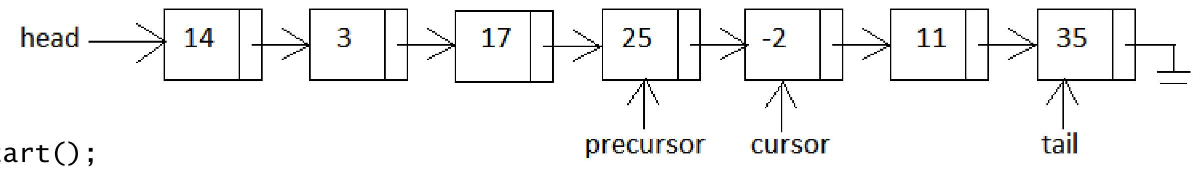
For each of the following scenarios, show how the reference fields of the DoubleLinkedSeq object change (head, tail, cursor, and precursor). Each scenario shows the sequence as it is before the code. You make changes to show how it will be after the code.



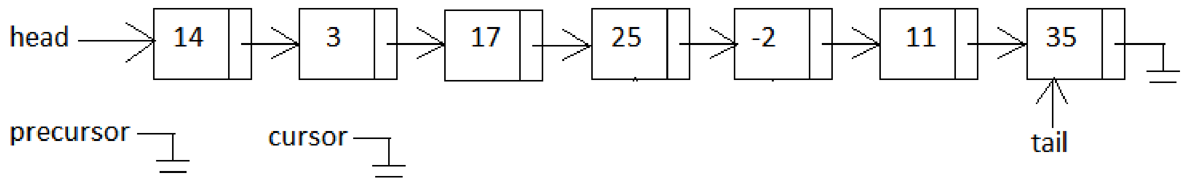
```
seq.advance();
seq.advance();
seq.advance();
```



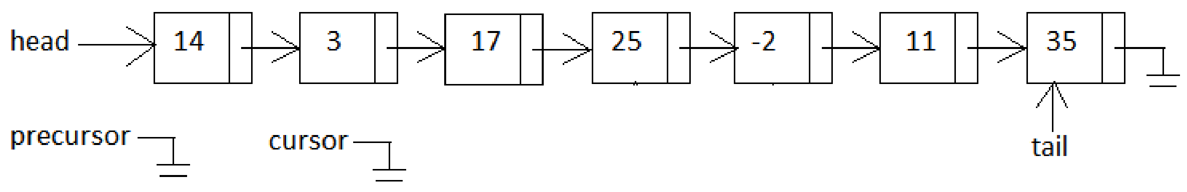
```
seq.removeCurrent();
```



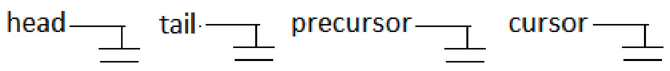
```
seq.start();
```



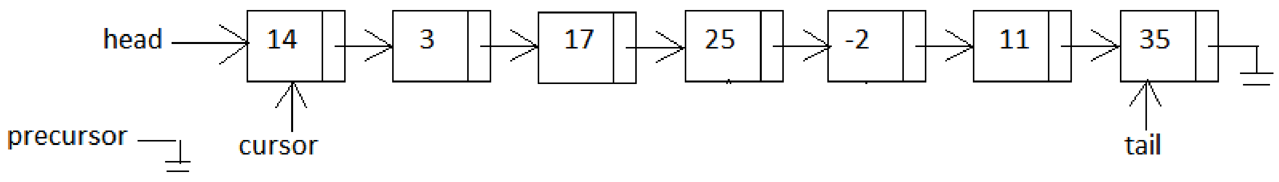
```
seq.addAfter(6);
```



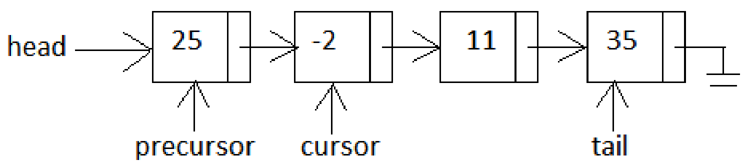
```
seq.addBefore(6);
```



```
seq.addAfter(14);
```



```
seq.addAfter(30);
```



```
seq.addAll(seq);
```

Homework: Generics

Name______________________________

1. Write a generic method called findElement that receives an array and an element. The method should return the index at which the first occurrence of the element is found, or -1 if the element is not found in the array.

2. Suppose we want to write a generic findMax routine. Consider the code below. This code cannot work because the compiler cannot prove that the call to compareTo is valid; compareTo is guaranteed to exist only if T is Comparable. Rewrite the header of the method so that it will work if T or any of its superclasses implemented the Comparable interface.

____

```
public static <T> T findMax(T[] a)
{
    int maxIndex = 0;
    for (int i = 1; i < a.length; i++)
    {
        if (a[i].compareTo(a[maxIndex]) > 0)
        {
            maxIndex = i;
        }
    }
    return a[maxIndex];
}
```

3. Make any changes necessary to the header, fields, and constructor to turn this part of the DoubleNode class into a generic Node class.

```
public class DoubleNode
{
    private double data;
    private DoubleNode link;


    public DoubleNode(double initialData, DoubleNode initialLink) {
        data = initialData;
        link = initialLink;
    }

    public DoubleNode() {
        data = 0.0;
        link = null;
    }
```

4. Make any changes necessary to the listCopyWithTail method for the generic Node class.

```
public static DoubleNode[] listCopyWithTail(DoubleNode source)
{
    DoubleNode[] answer = new DoubleNode[2];
    // Handle the special case of the empty list.
    if (source != null)
    {
        // Make the first node for the newly created list.
        DoubleNode copyHead = new DoubleNode(source.data, null);
        DoubleNode copyTail = copyHead;

        // Make the rest of the nodes for the newly created list.
        while (source.link != null)
        {
            source = source.link;
            copyTail.addNodeAfter(source.data);
            copyTail = copyTail.link;
        }

        // Return the head and tail references.
        answer[0] = copyHead;
        answer[1] = copyTail;
    }

    return answer;
 }
```

5. Suppose we are changing the IntLinkedBag to a generic LinkedBag. Indicate the necessary changes that we would need to make to the following equals method. Assume the generic Node class.

```
public boolean equals(Object other)
{
    if (this == other)
        return true;
    if (other instanceof IntLinkedBag)
    {
        IntLinkedBag otherBag = (IntLinkedBag) other;
        if (manyNodes != otherBag.manyNodes)
            return false;
        for (IntNode t1 = head, IntNode t2 = otherBag.head; t1 != null;
             t1 = t1.getLink(),  t2 = t2.getLink())
        {
            if (t1.getData() != t2.getData())
                return false;

        }
        return true;
    }
    return false;
}
```

Homework: Stack Applications

Name ______________________________

1. Put the following infix expressions into postfix notation:

   a. 23 – 14 + ((12 + 6 * 3) / (4 * 2) – 5

   ____

   b. (4 * a + a) * a / (b + 10)

   ____

   c. 22 + 8 * 14 / (33 – 12 * 3)

   ____

2. Evaluate the following postfix expressions (the answer is a number):

   a. 60 6 12 + 3 * 2 + 8 /

   ____

   b. 22 8 12 3 / - 10 + 9 - + 3 /

   ____

   c. 10 20 + 16 8 / * 40 + 2 /

   ____

Homework: Queues

Name______________________________

1. Consider the following ArrayList implementation of a queue:

```
public class ArrayListQueue<E> {
    private ArrayList queue;
    public ArrayListQueue() { queue = new ArrayList<E>(); }
    public E dequeue(){ return queue.remove(0); }
    public void enqueue(E element) { queue.add(element); }
    public E peek() { return queue.get(0); }
}
```

   Why is this implementation inefficient? Would it be better to make the front of the queue be at the end of the ArrayList and the rear of the queue be at the front of the ArrayList?

2. A queue can also be implemented using two stacks. One stack is used to enqueue elements, while the other is used to dequeue elements. When an element is added to the queue it is pushed on the **in** stack. When an element is removed from the queue it is popped off the **out** stack. If the **out** stack is empty when an element needs to be removed, the elements of the **in** stack are transferred to the **out** stack and then the element is removed. This is a common implementation of a queue for functional programming languages like Haskell.

   Draw two stacks and use them to carry out the following operations on a queue: add(1), add(2), add(3), remove(), add(4), remove(), remove(), add(5), add(6), remove(), add(7), add(8), add(9), remove(). Clearly show the final contents of each stack.

Homework: Recursion

Name________________________________

1. This method is called with gretel(20). What is returned?

```
public int gretel (int num)
{
    int x;
    if (num < 12)
    {
        x = 12;
    }
    else
    {
        x = num * gretel(num-5);
    }
    return x;
}
```

2. What is the output from hansel(4)?

```
public void hansel (int num)
{
    if (num > 0)
    {
        System.out.println("Line 5: " + num);
        hansel(num-1);
        System.out.println("Line 7: " + num);
    }
}
```

3. What is the output of mouse(3)?

```
public void mouse(int num)
{
    if (num > 0)
     {
        System.out.println( "It is April.");
        mouse(num-1);
    }
    System.out.println("Almost over " + num + "!");
}
```

4. What is returned by woods(3)?

```
public int woods(int num)
{
    int x;
    if (num <= 0)
    {
       x = num;
    }
    else if (num % 2 == 0)
    {
       x = woods(num-1) + 4;
    }
    else
    {
       x = woods(num-1) + woods(num-2);
    }
    return x;
}
```

Homework: Quadratic Sorting Algorithms

Name______________________________

1. Here is a chart showing how the elements of an array change as the selection sort algorithm is used to sort the array. At each step the smallest element of the array is found and is then swapped into its appropriate position. The unsorted portion of the array then shrinks by one element. Make a similar chart for the second array. If at some step the row doesn't change, you should still rewrite the row.

25	7	3	9	12	1	20	26	5	14
1	7	3	9	12	25	20	26	5	14
1	3	7	9	12	25	20	26	5	14
1	3	5	9	12	25	20	26	7	14
1	3	5	7	12	25	20	26	9	14
1	3	5	7	9	25	20	26	12	14
1	3	5	7	9	12	20	26	25	14
1	3	5	7	9	12	14	26	25	20
1	3	5	7	9	12	14	20	25	26
1	3	5	7	9	12	14	20	25	26

14	82	11	35	99	7	3	75	24	9

2. Here is a chart showing how the elements of the array change during insertion sort. Make a similar chart for the second array. If at some step the row doesn't change, you should still rewrite the row.

25	7	3	9	12	1	20	26	5	14
7	25	3	9	12	1	20	26	5	14
3	7	25	9	12	1	20	26	5	14
3	7	9	25	12	1	20	26	5	14
3	7	9	12	25	1	20	26	5	14
1	3	7	9	12	25	20	26	5	14
1	3	7	9	12	20	25	26	5	14
1	3	7	9	12	20	25	26	5	14
1	3	5	7	9	12	20	25	26	14
1	3	5	7	9	12	14	20	25	26

14	82	11	35	99	7	3	75	24	9

3. Now show how the array changes after each pass of bubble sort. At each step, bubble the smallest element leftward. You don't have to record all the swaps – show the data at the end of each pass.

14	82	11	35	99	7	3	75	24	9

Homework: Quicksort

Name________________________

1. Run the partition code provided on the handout from class on this array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9	6	15	9	9	18	9	18	3	15	9	6	3	9	6	9	9	3	15	9

mid = _____ pivot = ______ pivot ends up in cell _____

2. Now imagine that the code inside the for loop is changed as follows (< in first while is changed to <=). With this change, all duplicates of the pivot end up in the left portion of the array. Run the partition code on the array again, using the changed code.

```
while (data[tooBigIndex] <= pivot)
    tooBigIndex++;
while (data[tooSmallIndex] > pivot)
    tooSmallIndex--;
if (tooSmallIndex <= tooBigIndex)
    break;
swap (data, tooBigIndex, tooSmallIndex);
tooBigIndex++;
tooSmallIndex--;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9	6	15	9	9	18	9	18	3	15	9	6	3	9	6	9	9	3	15	9

mid = _____ pivot = ______ pivot ends up in cell ______

3. If there are many duplicates in your data, the version of partition given in class is best. Why?