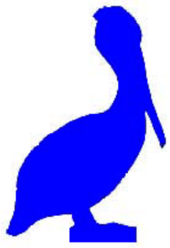
# Blue Pelican Java



by Charles E. Cook

Version 7.0.1A

“Blue Pelican Java,” by Charles E. Cook. ISBN 1-58939-758-4.

# Preface

You will find this book to be somewhat unusual. Most computer science texts will begin with a section on the history of computers and then with a flurry of definitions that are just "so many words" to the average student. My approach with Blue Pelican Java is to first give the student some experience upon which to hang the definitions that come later, and consequently, make them more meaningful.

This book does have a history section in Appendix S and plenty of definitions later when the student is ready for them. If you will look at Lesson 1, you will see that we go right to work and write a program the very first day. The student will not understand several things about that first program, yet he can immediately make the computer do something useful. This work ethic is typical of the remainder of the book. Rest assured that full understanding comes in time. Abraham Lincoln himself subscribed to this philosophy when he said, "Stop petting the mule, and load the wagon."

The usual practice in most Java textbooks of introducing classes and objects alongside the fundamental concepts of primitive variable types, loops, decision structures, etc. is deferred until the student has a firm grasp of the fundamentals. Thus, the student is not overwhelmed by **simultaneous** introduction of OOPs (Object Oriented Programming) and the fundamentals. Once introduced, (Lesson 15), OOPs is heavily emphasized for the remainder of the book.

I fully realize that there are those who disagree with this idea of deferring the introduction of OOPs, and from their own point of view, they are right. In most cases they teach only the very highest achieving, mature students. In those cases, I agree that it is acceptable to begin with OOPs; however, for the average student and especially for younger high school students, I feel that they need to understand the fundamentals first.

Upon first examination of this book it may not appear to be very "robust" in that there is not great depth for some of the topics. Actually the depth **is** there,… in the Appendix. The Appendix for this book is unusually large. Here is why the book is organized this way:

- The lessons are kept purposely short so as to hold down the intimidation factor. As a result, the lessons should look "doable" to the students.
- The in-depth material is placed in the Appendices, and references to the Appendices are made in the lessons. As an example, in Lesson 18 the *split* method is introduced. The *split* method uses regular expressions that are briefly discussed there; however, the in-depth presentation of regular expressions is placed in Appendix AC.

Unfortunately, this book does not introduce any graphics or windows programming. The 57 lessons in this book can be covered in one school year, but just barely. To prepare students for the AP test (and contests) there is only time to cover the essentials presented in this book. Check http://www.bluepelicanjava.com for the availability of study materials for the current **AP case study**, updates on this book, **videos** for each lesson, and an inexpensive way to purchase hard-cover books.

I am often asked **how to use this book**. "Which lessons are really important and which can be skipped?" The answer is simple:

- **Start on Lesson 1.**
- **Proceed at a reasonable rate.** (See Appendix P for a time-line.)
- **Don't skip anything** (except for, perhaps Lesson 47 and Lesson 53)
- **Give a simple, confidence-building quiz on each lesson.** Quizzes and keys are provided in the Answer Book (available at www.bluepelicanjava.com).
- **Make sure the students do the provided exercises and projects.**
- **Give tests at regular intervals.** Tests and keys are provided in the Answer Book.

In this book you will also notice another part of my philosophy of teaching and educational material in general…**Keep it simple**… I try to keep things as simple and uncluttered as possible. For example, you will find specific examples in greater numbers than long-winded explanations in this book. You won't find many pictures and sidebars and lots of little colored side notes scattered about. Some of that type format does contain some useful information; however, I feel that it is largely distracting. Apparently more and more people are coming around to my way of thinking on this, and here is why I think so. Recall that just a few years ago that nearly all web pages looked like cobbled together ransom notes with just a profusion of colors, links, and tidbits scattered all over the page. Take a look at professional web pages today. They typically have a very neat, clean appearance…often with just a plain white background and with plenty of space between the various elements. This is good. Simple is better.

Since this textbook has a strong emphasis on preparation for the AP test and competition (computer science contests), special "contest type" problems are provided at the end of most lessons. I realize that most students will not compete and some may not even take the AP exam; however, the material is not wasted on them. Those "contest type" problems are good for the average student too, as long as they are not overwhelmed with too many problems at one sitting. Hopefully, I have just the optimum number of these type problems on each lesson and students won't be burned-out by too much of a good thing.

Finally, we come to the reason for the choice of Blue Pelican Java as a name for this book. One of the early (and free) java IDE's available for students was BlueJ and it was the first my students used. I always thought BlueJ was an elegant name and had expressed a desire to a colleague to continue the tradition by naming the book after some other blue-colored bird. He jokingly suggested Blue Pelican, not really being serious about naming a book after this rather ungainly, clunky bird. For the lack of an existing name for the book during development, it continued to be called Blue Pelican. If you call something by a particular name long enough, that's its name, and so the name stuck.

I truly hope Blue Pelican Java is useful to you and that you find the experience of learning to program a rewarding one. Just remember, few things worthwhile are acquired without some sacrifice. The "sacrifice" here will be the time you invest in creating programs and trying the code suggested in these pages.

Charles E. Cook

# Table of Contents

Some of the numbered lessons below are marked with an asterisk (*). This indicates they are subjects not covered by the AP A test. All other lessons have at least "potential relevance".

# Lesson 11…..The *for*-Loop

One of the most important structures in Java is the "*for*-loop". A loop is basically a block of code that is **repeated** with certain rules about how to start and how to end the process.

**Simple example:**

Suppose we want to sum up all the integers from 3 to 79. One of the statements that will help us do this is:

```
sum = sum + j;
```

However, this only works if we repeatedly execute this line of code, …first with *j = 3*, then with *j = 4, j = 5*, …and finally with *j = 79*. The full structure of the *for*-loop that will do this is:

```
int j = 0, sum = 0;
for (j = 3; j <= 79; j++)
{
        sum = sum + j;
        System.out.println(sum);  //Show the progress as we iterate thru the loop.
}

System.out.println("The final sum is " + sum); // prints 3157
```

**Three major parts:**

Now let's examine the three parts in the parenthesis of the *for*-loop.

**<u>Initializing expression</u>**…*j = 3* If we had wanted to start summing at 19, this part would have read, *j = 19*.

**<u>Control expression</u>**…*j <= 79* We continue looping as long as this *boolean* expression **is *true***. In general this expression can be **any** *boolean* expression. For example, it could be:

```
count = = 3          s + 1 < alphB        s > m +19          etc.
```

**Warning:** There is something really bad that can happen here. You must write your code so as to insure that this control statement will eventually become *false*, thus causing the loop to terminate. Otherwise you will have an <u>endless loop</u> which is about the worst thing there is in programming.

**<u>Step expression</u>**… *j++* This tells us how our variable changes as we proceed through the loop. In this case we are incrementing *j* each time; however, other possibilities are:

```
j--          j = j + 4          j = j * 3          etc.
```

For our example above, exactly when does the increment …*j++* occur? Think of the step expression being at the <u>bottom</u> of the loop as follows:

```
for (j = 3; j <= 79; . . . )
{
        … some code …

        j++;  //Just think of the j++ as being the last line of code inside the
}             //braces.
```

**Special features of the for-loop:**

**The *break* command:**

If the keyword ***break*** is executed inside a *for*-loop, the loop is immediately exited (regardless of the control statement). Execution continues with the statement immediately following the closing brace of the *for*-loop.

**Declaring the loop variable:**

It is possible to declare the loop variable in the initializing portion of the parenthesis of a *for*-loop as follows:

```
for (int j = 3; j <= 79; j++)
{
        . . .
}
```

In this case the **scope** of *j* is confined to the interior of the loop. If we write *j* in statement outside the loop (without redeclaring it to be an *int*), it won't compile. The same is true of any other variable declared inside the loop. Its scope is limited to the interior of the loop and is not recognized outside the loop as is illustrated in the following code:

```
for (j = 3; j <= 79; j++)
{
        double d = 102.34;

        . . .
}
System.out.println(d);  //won't compile because of this line
```

**No braces:**

If there is only **one line of code** or just one basic structure (an *if*-structure or another loop) inside a loop, then the braces are unnecessary. In this case it is still correct (and highly recommended) to still have the braces…but you **can** leave them off.

```
for (j = 3; j <= 79; j++)          is equivalent to          for (j = 3; j <= 79; j++)
     sum = sum + j;                                          {  sum = sum + j; }
```

**When the loop finishes:**

It is often useful to know what the loop variable is after the loop finishes:

```
for (j = 3; j <= 79; j++)
{
        . . . some code . . .
}
System.out.println(j);  //80
```

On the last iteration of the loop, *j* increments up to 80 and this is when the control statement *j <= 79* finally is *false*. Thus, the loop is exited.

**Nested loops:**

"Nested loops" is the term used when one loop is placed inside another as in the following example:

```
for(int j = 0; j < 5; j++)
{
        System.out.println("Outer loop");  // executes 5 times
        for(int k = 0; k < 8; k++)
        {
                System.out.println("...Inner loop");  //  executes 40 times
        }
}
```

The inner loop iterates eight times for **each** of the five iterations of the outer loop. Therefore, the code inside the inner loop will execute 40 times.

******************************************************************

**Warning:**

A very common mistake is to put a semicolon immediately after the parenthesis of a *for*-loop as is illustrated by the following code:

```
for (j =3; j <= 79; j++);
{
        //This block of code is only executed once because of the inappropriately
        //placed semicolon above.
        . . . some code . . .
}
```

## Exercise for Lesson 11

In each problem below state what is printed unless directed otherwise.

1. 
```
int j = 0;
for (int g = 0; g <5; g++)
        j++;
System.out.println(j);
```

2. 
```
int s = 1;
for (int j = 3; j >= 0; j--)
{
        s = s + j;
}
System.out.println(s);
```

3. 
```
int p = 6;
int m = 20, j;
for (j = 1; j < p; j++);  //Notice the semicolon on this line
{
        m = m + j * j;
}
System.out.println(m);
```

4. 
```
double a = 1.0;
for (int j = 0; j < 9; j++)
{
        a*=3;
}
System.out.println(j);
```

5. 
```
for (int iMus = 0; iMus < 10; iMus++)
{
        int b = 19 + iMus;
}
System.out.println(b);
```

6. 
```
double d = 100.01;
int b = 0;
for (int iMus = 0; iMus < 10; iMus++)
        b = 19 + iMus;
        d++;
System.out.println(d);
```

7. Write a for-loop that will print the numbers 3, 6, 12, and 24

8. Write a for-loop that will print the numbers 24, 12, 6, 3

9. int k = 0;
```
for(int j = 0; j <= 10; j++)
{
        if (j = = 5)
        {
                break;
        }
        else
        {
                k++;
        }
}
System.out.println(k);
```

10. What is the name of the part of the parenthesis of a for-loop that terminates the loop?

11. What is the value of *j* for each iteration of the following loop?
```
int i, j;
for( i = 10; i <= 100; i = i+ 10)
        j = i / 2;
```

12. What is the value of *r* after the following statements have executed?
```
int r, j;
for (j = 1; j < 10; j = j * 2)
r = 2 * j;
```

13. What is the worst sin you can commit with a for-loop (or any loop for that matter)?

14. How many times does the following loop iterate?
```
for (p = 9; p <= 145; p++)
{
        . . .
}
```

## Project… Name Reversal

Write a program that will allow a user to input his name. The prompt and input data would look something like this:

```
Please enter your name. Peter Ustinov
```

Using a for-loop and the *String* method, *substring(...),* produce a printout of the reversal of the name.

For example, the name *Peter Ustinov* would be:

```
vonitsu retep
```

Notice that the printout is in all lower-case. Use the *String* method, *toLowerCase( )* to accomplish this.

## *for*-Loop… Contest Type Problems

1. What is output?

   A. 0
   B. 10
   C. 15
   D. 5
   E. None of these

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

2. What is output?

   A. 66
   B. 100
   C. 101
   D. 99
   E. None of these

```
double kk = 3;
int j = 0;
for( j = 0; j <= 100; j++)  {
        kk = kk + Math.pow(j, 2);
        ++kk;
}
System.out.println(j);
```

3. What is the final value of p?

   A. 10
   B. 4
   C. 5
   D. 12
   E. None of these

```
double p = 0;
for(int m =10; m > 6; --m)
{
        if(m= =7)  {
                p = p+m;
        }
        else  {
                ++p;
        }
}
```

4. Which of the following will print the set of odd integers starting at 1 and ending at 9?

   A. for(int j=0; j<=9; j++) { System.out.println(j); }
   B. for(int j=1; j<10; j= j+2) { System.out.println(j); }
   C. for(int j=1; j<=9; j+=1) { System.out.println(j); }
   D. for(int j=1; j<=9; j+=2) { System.out.println(j); }
   E. Both B and D

5. What is output?

   A. 4950
   B. 101
   C. 100
   D. Nothing, it's an endless loop
   E. None of these

```
double x = 0;
for(int b=0; b<101; b++)
{
        x = x + 1;
        b--;
}
System.out.println(x);
```

6. What is output?

   A. 5   6
   B. 6   6
   C. 5   10
   D. 5   5
   E. None of these

```
int p, q=5;
for(p=0; p<5; p++);  //notice the semicolon
        q = q+1;
System.out.println(p + "   " + q);
```

| | |
|---|---|
| 7. What is output?<br><br>A. 98<br>B. 3939<br>C. 109<br>D. 4039<br>E. None of these | <pre>int j, k;<br>int count = 0;<br>for(j=0; j<4; j++)<br>{<br>        for( k = 0; k < 10; k++ )<br>        {<br>                count++;<br>        }<br>}<br>System.out.print(count--);<br>System.out.println(count);</pre> |

# Lesson 14…..Binary, Hex, and Octal

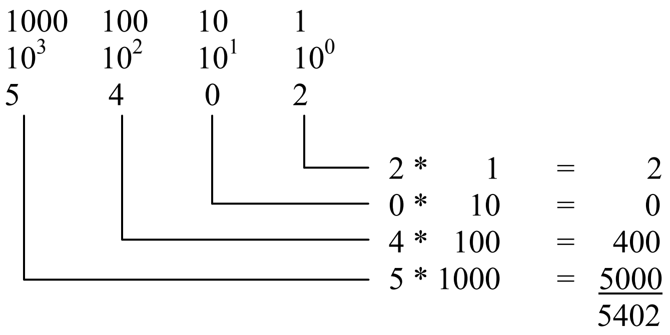
We will examine four different number systems here,…decimal, binary, hexadecimal (hex), and octal. In your study of these number systems it is very important to note the **similarities** of each. Study these similarities carefully. This is how you will understand the new number systems.

**Decimal, base 10**

There are only **10** digits in this system:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Note that even though this is base **10**, there is no single digit for **10**. Instead we use two of the permissible digits, 1 and 0 to make **10**.

Positional value: Consider the decimal number 5,402.

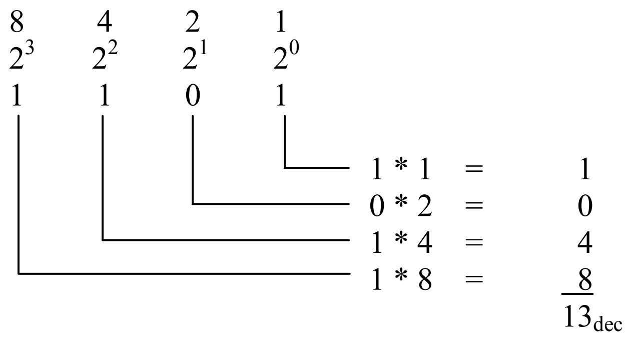


**Binary, base 2**

There are only **2** digits in this system:
0, 1

Note that even though this is base **2**, there is no single digit for **2**. Instead we use two of the permissible digits, 1 and 0 to make **$10_{bin}$ ($2_{dec}$)**.

Positional value: Consider the conversion of binary number $1101_{bin}$ to decimal form.



Bits and Bytes: Each of the positions of $1101_{bin}$ is called a bit… it's a four-bit number. When we have **eight bits** (example, $10110101_{bin}$) we call it a **byte**. If we say that a computer has 256mb of memory, where *mb* stands for megabytes, this means it has 256 million bytes. See Appendix Y for more on kilobytes, megabytes, and gigabytes, etc.

**Hexadecimal (hex), base 16**

There are only **16** digits in this system:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 10 | 11 | 12 | 13 | 14 | 15 |

Note that even though this is base **16**, there is no single digit for **16**. Instead we use two of the permissible digits, 1 and 0 to make **$10_{hex}$ ($16_{dec}$)**.

<u>Positional value</u>: Consider the conversion of hex number $5C02_{hex}$ to decimal form.

4096  256  16  1

$16^3$  $16^2$  $16^1$  $16^0$

5  C  0  2

2 * 1 = 2

0 * 16 = 0

12 * 256 = 3072

5 * 4096 = <u>20480</u>

$23554_{dec}$

**Octal, base 8**

There are only **8** digits in this system:

0, 1, 2, 3, 4, 5, 6, 7

Note that even though this is base **8**, there is no single digit for **8**. Instead we use two of the permissible digits, 1 and 0 to make **$10_{oct}$ ($8_{dec}$)**.

<u>Positional value</u>: Consider the conversion of octal number $5402_{oct}$ to decimal form.

512  64  8  1

$8^3$  $8^2$  $8^1$  $8^0$

5  4  0  2

2 * 1 = 2

0 * 8 = 0

4 * 64 = 256

5 * 512 = <u>2560</u>

$2818_{dec}$

Following are examples that show how we can use these different number systems with Java.

**Store a hex number:**

```
int x = 0x4CB3; //the leading 0x indicates hex format
System.out.println(x); //19635 …Notice it automatically prints in decimal form
```

**Store an octal number:**

```
int x = 0734; //the leading 0 indicates octal format
System.out.println(x); //476 …Notice it automatically prints in decimal form
```

**Convert an integer variable to a hex *String*:**

```
int x = 3901;
System.out.println( Integer.toHexString(x) );  //f3dhex
                    //…or Integer.toString(x, 16);
```

**Convert an integer variable to a binary *String*:**

```
int x = 3901;
System.out.println( Integer.toBinaryString(x) );  // 111100111101bin
        //…or Integer.toString(x, 2);
```

**Convert an integer variable to an octal *String*:**

```
int x = 3901;
System.out.println( Integer.toOctalString(x) );  //7475oct
                    //…or Integer.toString(x, 8);
```

Notice in the last three examples above the following method was an alternate way to convert bases:

```
String s = Integer.toString(i, b);
```

The first parameter *i* is of type *int* and *b* is the base to which we wish to convert. *b* (also an *int* type) can be any base ranging from 2 to 36. Just as for hexadecimal numbers where we use the letters A – F, the bases higher than 16 (hex) use the remaining letters of the alphabet. For example, Integer.toString(8162289, 32) returns "7p2vh".

**Base conversion using *parseInt*:**

It is also possible to go from some strange base (in *String* form) back to a decimal *int* type. For example, *Integer.parseInt("3w4br", 35)* converts $3w4br_{35}$ into $5879187_{dec}$.

**A technique for converting 147 from decimal to binary:**

```
2  147              ; 2 divides into 147   73 times with a remainder of
2   73  1           ;1.  2 divides into 73, 36 times with a remainder of
2   36  1           ;1.  2 divides into 36, 18 times with a remainder of
2   18  0           ;0.  2 divides into 18, 9 times with a remainder of
2    9  0           ;0.  etc.
2    4  1
2    2  0
2    1  0
     0  1       Now list the 1's and 0's from bottom to top. 10010011bin = 147dec
```

**A technique for converting 3741 from decimal to hex:**

```
16     3741         ;divide 3741 by 16. It goes 233 times with a
16     233    13    ;remainder of 13.
16     14     9
       0      14
```

Now list the numbers from <u>bottom to top</u>. Notice, when listing the 14 we give its hex equivalent, E, and for 13 we will give D: $E9D_{hex} = 3741_{dec}$

**An octal multiplication example ($47_{oct}$ * $23_{oct}$):**

| | | |
|---|---|---|
| 2 | | |
| 47 | $3 * 7 = 21_{dec}$ | 8 divides into 21  2 times with a remainder |
| 23 | | of 5. Notice the 5 and the carry of 2 |
| 5 | | |

| | | |
|---|---|---|
| 1 2 | | |
| 47 | $(3*4) + 2 = 14_{dec}$ | 8 divides into 14  1 time with a |
| 23 | | remainder of 6 |
| 165 | | |

| | | |
|---|---|---|
| 1 | | |
| 47 | $2 * 7 = 14_{dec}$ | 8 divides into 14  1 time with a |
| 23 | | remainder of 6 |
| 165 | | |
| 6 | | |

| | | |
|---|---|---|
| 11 | | |
| 47 | $2*4 + 1 = 9_{dec}$ | 8 divides into 9  1 time with a |
| 23 | | remainder of 1 |
| 165 | | |
| 116 | | Now we are ready to add: |

| | | |
|---|---|---|
| 1 | | |
| 165 | | |
| 116 | | |
| $1345_{oct}$ | Notice in adding 6 + 6  we get 12.  8 divides 12   1 | |
| | | time, remainder 4. |

**Binary addition:**

The rules to remember are:   0 + 0 = 0     0 + 1 = 1     1 + 1 = 0 with a carry of 1

Add the two binary numbers 110011  and 100111.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | | | 1 | 1 | 1 | |
| | 1 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 |

The problem we have done here:   $110011_{bin} + 100111_{bin} = 1011010_{bin}$

is equivalent to:  $51_{dec} + 39_{dec} = 90_{dec}$

**A trick for converting binary into hex:**

Begin with the binary number 10110111010. Starting on the right side, partition this into groups of four bits and get   101  1011  1010  To each four bit group, assign a hex digit.
5   B   A

Thus we have $10110111010_{bin} = 5BA_{hex}$. Similarly, partition a binary number into groups of 3 to convert to Octal.

**See <u>Appendix D</u> for the decimal, hex, octal, and binary equivalents of 0 – 127.**

For an enrichment activity concerning a **Binary** File Editor, see Appendix U. There, you will have an opportunity to specify software, search on the Internet, and publish the information you discover…. Appendix G explains how negative numbers are handled in the binary system.

## Project… Basically Speaking

Create a project called *TableOfBases* with class *Tester*. The *main* method should have a *for* loop that cycles through the integer values 65 <= j <= 90 (These are the ASCII codes for characters A – Z). Use the methods learned in this lesson to produce a line of this table on each pass through the loop. Display the equivalent of the decimal number in the various bases just learned (binary, octal, and hex) as well as the character itself:

| Decimal | Binary | Octal | Hex | Character |
|---|---|---|---|---|
| 65 | 1000001 | 101 | 41 | A |
| 66 | 1000010 | 102 | 42 | B |
| 67 | 1000011 | 103 | 43 | C |
| 68 | 1000100 | 104 | 44 | D |
| 69 | 1000101 | 105 | 45 | E |
| 70 | 1000110 | 106 | 46 | F |
| 71 | 1000111 | 107 | 47 | G |
| 72 | 1001000 | 110 | 48 | H |
| 73 | 1001001 | 111 | 49 | I |
| 74 | 1001010 | 112 | 4a | J |
| 75 | 1001011 | 113 | 4b | K |
| 76 | 1001100 | 114 | 4c | L |
| 77 | 1001101 | 115 | 4d | M |
| 78 | 1001110 | 116 | 4e | N |
| 79 | 1001111 | 117 | 4f | O |
| 80 | 1010000 | 120 | 50 | P |
| 81 | 1010001 | 121 | 51 | Q |
| 82 | 1010010 | 122 | 52 | R |
| 83 | 1010011 | 123 | 53 | S |
| 84 | 1010100 | 124 | 54 | T |
| 85 | 1010101 | 125 | 55 | U |
| 86 | 1010110 | 126 | 56 | V |
| 87 | 1010111 | 127 | 57 | W |
| 88 | 1011000 | 130 | 58 | X |
| 89 | 1011001 | 131 | 59 | Y |
| 90 | 1011010 | 132 | 5a | Z |

## Exercise on Lesson 14

1. Convert $3C4F_{hex}$ to decimal.

2. Convert $100011_{bin}$ to decimal.

3. Convert $637_{oct}$ to decimal.

4. Is the following code legal? If not, why? `int v = 04923;`

5. Is the following code legal? If not, why? `int w = 0xAAFF;`

6. Convert $9A4E_{hex}$ to decimal.
7. Convert $1011011_{bin}$ to decimal.
8. Convert $6437_{oct}$ to decimal.
9. Write code that will store $5C3B_{hex}$ in the integer variable *a*.
10. Write code that will store $3365_{oct}$ in the integer variable *k*.
11. Convert $478_{dec}$ to binary.
12. Convert $5678_{dec}$ to hex.
13. Convert $5678_{dec}$ to octal.
14. Multiply $2C6_{hex}$ times $3F_{hex}$ and give the answer in hex.
15. Add $3456_{oct}$ and $745_{oct}$ and give the answer in octal.
16. What is the decimal equivalent of $A_{hex}$?
17. What is the decimal equivalent of $8_{hex}$?
18. What is the base of the hex system?
19. How do you write $16_{dec}$ in hex?
20. What is the base of the binary system?
21. Add these two binary numbers: 1111000 and 1001110.
22. Add these two binary numbers: 1000001 and 1100001
23. Explain the following “joke”: “There are only 10 types of people in the world…those who understand binary and those who don’t.”
24. Suppose you have *String s* that represents a number that you know is expressed in a base given by *int b*. Write code that will convert this into an equivalent decimal based integer and store the result in *int i*.
25. Show code that will convert $9322gf_{33}$ into *String s* that is the equivalent in base 28.
26. Add $3FA6_{hex}$ to $E83A_{hex}$ and give the answer in hex.
27. Multiply $7267_{oct}$ times $4645_{oct}$ and give the answer in octal.
28. Add $2376_{oct}$ to $567_{oct}$ and give the answer in octal.
29. Multiply $3E_{hex}$ times $5B_{hex}$ and give the answer in hex.

# Lesson 29…..Random Numbers

**Why random?**

Why would we want random numbers? What possible use could there be for the generation of **unpredictable** numbers? It turns out there are plenty of applications, and the following list suggests just a few:

1. Predictions for life expectancy …used in insurance
2. Business simulations
3. Games …gives users a different experience each time
4. Simulations for scientific research, etc.

**Important methods:**

The *Random* class (requires the import of *java.util.Random*) generates random numbers and has three methods, besides the constructor, that are of interest to us. These are not *static* methods, so we must first create an object:

**Constructor**

public Random( ) // **Signature**

**Example:**

Random rndm = new Random( );

***nextInt( )***

public int nextInt( ) // **Signature**

This yields a randomly selected integer in the range Integer.MIN_VALUE to Integer.MAX_VALUE. (-2,147,843,648 to 2,147,843,647 as specified in Appendix C).

**Example:**

```
int x = rndm.nextInt( ); //x could be any integer from -2,147,843,648 to
                               //2,147,843,647
```

***nextInt(n)***

public int nextInt(int n) // **Signature**

This yields a randomly selected integer (0, 1, 2, …, n-1).

**Example:**

int x = rndm.nextInt(21); //x could be any integer from 0 to 20, inclusive for both

***nextDouble( )***

public double nextDouble( ) // **Signature**

This yields a randomly selected *double* from 0 (inclusive) to 1 (exclusive) and behaves exactly as does Math.random( ) (discussed in Lesson 6).

**Example:**

double d = rndm.nextDouble( ); //generates doubles in the range $0 \leq d < 1$

Because of the two versions of *nextInt*, we notice that our *Random* class has two methods of the same name (but different parameters). We say the methods named *nextInt* are

**overloaded**. In some contexts overloading is bad (example, overloading a truck). However, in the software sense of overloading, it is perfectly normal and acceptable.

**Typical Problems:**

1. Suppose we want a range of integers from 90 to 110, inclusive for both.

   First we subtract (110 – 90 = 20). Then add 1 to get 21. Now set up your code as follows to generate the desired range of integers:

   ```
   int r = 90 + rndm.nextInt(21);
   ```

   Put this last line of code in a *for*-loop, and you will see a range of integers from 90 to 110. Loop through 1000 times, and likely you will see every value…most will be repeated several times.

   ```
   int r = 0, count = 0;
   Random rndm = new Random( );
   for(int j = 0; j < 1000; j++)  {
           r = 90 + rndm.nextInt(21);
           System.out.print(r + "   ");

           //For convenience in viewing on a console screen, the following loop
           //produces a new line after 15 numbers are printed side-by-side.
           count++;
           if(count >15) {
                   System.out.println("  ");
                   count = 0;
           }
   }
   ```

2. Suppose we wish to generate a continuous range of floating point numbers from 34.7838 (inclusive) to 187.056 (exclusive). How would we do this?
   First, subtract (187.056 – 34.7838 = 152.2722). Now set up your code as follows to generate the desired range.

   ```
   Random rndm = new Random( );
   double r;
   r = 34.7838 + 152.2722 * rndm.nextDouble( );
   // Generates continuous floating point numbers in the range
   // 34.7838   ≤   r   <  187.056
   ```

**Some additional methods of the *Random* class:**

*nextBoolean( )* … returns a random *boolean* value (*true* or *false*).

*nextGaussian( )* … returns a Gaussian ("normally") distributed *double* with a mean value of 0.0 and a standard deviation of 1.0.

## Project… Generate Random Integers

As described in problem 1 above, generate 33 random integers in the inclusive range from 71 to 99.

## Project… Generate Random Doubles

As described in problem 2 above, generate 27 random *double*s in the inclusive range from 99.78 to 147.22.

## Exercise on Lesson 30

In the following problems assume that *rndm* is an object created with the *Random* class. Assume *d* is of type *double* and that *j* is of type *int*.

1. What range of random numbers will this generate?
   j = 201 + rndm.nextInt(46);

2. What range of random numbers will this generate?
   d = 11 + 82.9 * rndm.nextDouble( );

3. What range of random numbers does *nextDouble( )* generate?

4. List all numbers that rnd*m.nextInt(10)* might generate.

5. Write code that will create an object called *rd* from the *Random* class.

6. Write code that will create a *Random* object and then use it to generate and print 20 floating point numbers in the continuous range $22.5 \le r < 32.5$

7. What import is necessary for the *Random* class?

8. Write code that will randomly generate numbers from the following set. Printout 10 such numbers.
   18, 19, 20, 21, 22, 23, 24, 25

9. Write code that will randomly generate and print 12 numbers from the following set.
   100, 125, 150, 175

10. Write a line of code to create a *Random* class object even though *Random* wasn't imported.

# Random Numbers… Contest Type Problems

1. Which of the following is a possible output?

   A. 0
   B. 36
   C. 37
   D. Throws an exception
   E. None of these

```
Random rd = new Random( );
System.out.println( rd.nextInt(36) );
```

2. To simulate the result of rolling a normal 6-sided die, what should replace **<*1>**

   A. rdm.nextDouble(6);
   B. rdm.nextInt(7);
   C. 1+ rdm.nextDouble(7);
   D. 1 + rdm.nextInt(6);
   E. 1 + rdm.nextDouble(6)

```
public static int dieOutcome( )
{
        Random rdm = new Random( );
        int die = <*1>
        return die;
}
```

3. Which of the following is a possible output of the code to the right?

   A. 0
   B. .9999
   C. 5.0
   D. 6.0
   E. None of these

```
java.util.Random rd = new java.util.Random( );
System.out.println( 1+ 5 * rd.nextDouble( ) );
```

4. What would be the range of possible values of *db* for the following line of code?
   double db = genRndDbl(4, 1);

   A. $1 \leq db < 5$
   B. $0 \leq db < 5$
   C. $1 \leq db < 4$
   D. $1 \leq db \leq 5$
   E. $0 \leq db \leq 5$

```
public static double genRndDbl(int m, int a)
{
    Random r = new Random( );
    double d = a + m * r.nextDouble( );
    return d;
}
```

5. What would be the replacement code for **<*1>** to generate random numbers from the following set?
   {20, 35, 50, 65}

   A. 20 * 15 + ri.nextInt(4);
   B. 20 + 15 * ri.nextInt(5);
   C. 15 * 20 + ri.nextInt(4);
   D. 15 + 20 * ri.nextInt(5);
   E. None of these

```
Random ri = new Random( );
int ri = <*1>
```

6. When a class has more than one method of the same name, this is called which of the following?

   A. overloading    B. inheritance    C. overriding    D. polymorphism

   E. None of these

7. Which of the following “tosses” a *Coin* object named *theCoin*, and produces a *true* when the *toss( )* method yields a *HEADS*?

A. theCoin.toss = = HEADS
B. toss = = 0
C. theCoin.toss( ) = = Coin.HEADS
D. theCoin.HEADS = = HEADS
E. Both C and D

8. Assuming that the *Random* class is “perfect” and generates all of the integers with equal probability, what is the probability that *toss( )* returns a head?

A. slightly over .5
B. slightly under .5
C. 1
D. exactly .5
E. None of these

```
public class Coin
{
    public Coin( )
    {
        r = new Random( );
    }

    public int toss( )
    {
        int i = r.nextInt( );
        if(i < 0)
        {
            return TAILS;
        }
        else
        {
            return HEADS;
        }
    }

    public static final int HEADS =  0;
    public static final int TAILS = 1;

    private Random r;
}
```

## Project… Monte Carlo Technique

Imagine a giant square painted outdoors, on the ground, with a painted circle inscribed in it. Next, image that it's raining and that we have the ability to monitor every raindrop that hits inside the square. Some of those raindrops will also fall inside the circle, and a few will fall in the corners and be inside the square, but not inside the circle. Keep a tally of the raindrops that hit inside the square (*sqrCount*) and those that also hit inside the circle (*cirCount*).

**The ratio of these two counts should equal the ratio of the areas** as follows: (Understanding this statement is essential. It is the very premise of this problem.)

$$\text{sqrCount} / \text{cirCount} = (\text{Area of square}) / (\text{Area of circle})$$

$$\text{sqrCount} / \text{cirCount} = \text{side}^2 / (\pi * r^2)$$

Solving for π from this equation we get

$$\pi = \text{cirCount} * (\text{side}^2) / (\text{sqrCount} * r^2)$$

So why did we solve for π? We already know that it's ≅ 3.14159. We simply want to illustrate that by a simulation (raindrop positions) we can solve for various things, in this case something we already know. The fact that we already know π just makes it that much easier to check our answer and verify the technique.

We are going to build a class called *MonteCarlo* in which the constructor will establish the size and position of our square and circle. Public state variables inside this class will be *h*, *k*, and *r*. These are enough to specify the position and size of our circle and square as shown in the figure to the right.

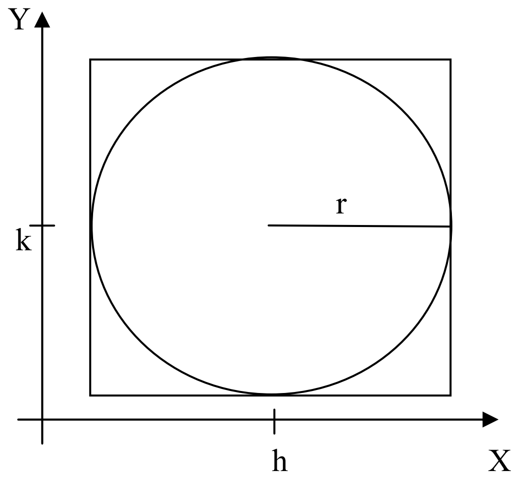


Fig. 30-1

**The requirements of your *MonteCarlo* class are:**

1. The constructor should receive *h*, *k*, and *r* as described above and use them to set the instance fields (state variables).

2. State variables *h*, *k*, and *r* are public *double*s. Create a *private* instance field as an object of the *Random* class. Call it *rndm*.

3. The *nextRainDrop_x( )* method should return a *double* that corresponds to a random raindrop's *x* position. The range of *x* values should be confined to the square shown above. No parameters are passed to this method.

4. The *nextRainDrop_y( )* method should return a *double* that corresponds to a random raindrop's *y* position. The range of *y* values should be confined to the square shown above. No parameters are passed to this method.

5. The method *insideCircle(double x, double y )* returns a *boolean*. A *true* is returned if the parameters *x* and *y* that are passed are either <u>inside</u> or <u>on</u> the circle.

   In writing this method, you must remember that the equation of a circle is
   $(x - h)^2 + (y - k)^2 = r^2$ …where *(h,k)* is the center and *r* is the radius.

   Also, the test for a point *(x, y)* being either <u>inside</u> or <u>on</u> a circle is
   $(x - h)^2 + (y - k)^2 <= r^2$

**You will need to build a *Tester* class with the following features:**

1. Class name, *Tester*

2. There is only one method, the *main( )* method.

3. Create a *MonteCarlo* object called *mcObj* in which the center of the circle is at (5, 3) and the radius is 2.

4. Set up a *for*-loop for 100 iterations:

5. Inside the loop obtain the random coordinates of a rain drop using the *nextRainDrop_x( )* and *nextRainDrop_y( )* methods.

6. Using the *x* and *y* just obtained, pass them as arguments to the *insideCircle( )* method to decide if our "raindrop" is inside the circle. If *insideCircle( )* returns a *true* then increment *cirCount*.

7. Increment *sqrCount* on each pass through the loop.

8. After the loop, calculate and print your estimate for $\pi$ according to the solution for $\pi$ on the previous page.

9. Change the number of iterations of the loop to 1000 and run the program again. Repeat for 10,000, 100,000, and 1,000,000 iterations. The estimate for $\pi$ should improve as the number of iterations increases.

# Lesson 39… Recursion

## What is recursion?

Software **recursion**, very simply, is the **process of a method calling itself**. This at first seems very baffling…somewhat like a snake swallowing its own tail. Would the snake eventually disappear?

## The classical factorial problem:

We will begin with the classical problem of finding the factorial of a number. First, let us define what is meant by "factorial". Three factorial is written as 3!, Four factorial is written as 4!, etc. But what, exactly, do they mean? Actually, the meaning is quite simple as the following demonstrates:

$$3! = 3 * 2 * 1 = 6$$
$$4! = 4 * 3 * 2 * 1 = 24$$

The only weird thing about factorials is that we define 0! = 1. There is nothing to "understand" about 0! = 1. It's a definition, so just accept it.

Here is an iterative approach to calculating 4!.

```
int answer = 1;
for(int j = 1; j <= 4; j++)
{
        answer = answer * j;
}
System.out.println(answer);  //24
```

Before we present the recursive way of calculating a factorial, we need to understand one more thing about factorials. Consider 6!.

$$6! = 6 * 5 * 4 * 3 * 2 * 1 = 6 * (5 * 4 * 3 * 2 * 1)$$

We recognize that the parenthesis could be rewritten as 5!, so 6! could be rewritten as

$$6! = 6 * (5!)$$

In general we can write n! = n(n –1)!. It is this formula that we will use in our recursive code as follows:

```
public static int factorial(int n)
{
        if(n = = 1)
        {    return 1;   }
        else
        {    return n * factorial(n – 1);  //notice we call factorial here    }
}
```

Call this code with System.out.println( factorial(4) );  //**24**

What really happens when the method calls itself? To understand this, we should pretend there are several copies of the *factorial* method. If it helps, you can think of the next one that is called as being named *factorial1*, and the next *factorial2*, etc. Actually, we need not pretend. This is very close to what really takes place. Analyzing the problem in this way, the last *factorial* method in this "chain" returns 1. The next to the last one returns 2, the next 3, and finally 4. These are all multiplied so the answer is 1 * 2 * 3 * 4 = 24.

**Short cuts:**

Let's look at some recursion examples using short cuts. For each problem, see if you can understand the pattern of how the answer (in bold print) was obtained.

1. System.out.println(adder(7)); // **46**

```
public static int adder(int n)
{
        if (n<=0)
                return 30;
        else
                return n + adder(n-2);
}
```

On the first call to *adder*, *n* is 7, and on the second call it's 5 (7 - 2), etc. Notice that in the *return* portion of the code that each *n* is **added** to the next one in the sequence of calls to *adder*. Finally, when the *n* parameter coming into *adder* gets to 0 or less, the returned value is 30. Thus, we have:

**7 + 5 + 3 + 1 + 30 = 46**

2. System.out.println(nertz(5)); // **120**

```
public static int nertz(int n)
{
        if (n = = 1)
                return 1;
        else
                return n * nertz(n-1);
}
```

On the first call to *nertz*, *n* is 5, and on the second call it's 4 (obtained with 5 - 1), etc. Notice that in the *return* portion of the code that each *n* is **multiplied** times the next one in the sequence of calls to *nertz*. Finally, when the *n* parameter coming into *adder* gets to 1, the returned value is 1. Thus, we have:

**5 * 4 * 3 * 2 * 1 = 120**

3. System.out.println(nrd(0)); // **25**

```
public static int nrd(int n)
{
        if (n > 6)
                return n - 3;
        else
                return n + nrd(n +1);
}
```

On the first call to *nrd*, *n* is 0, and on the second call it's 1 (obtained with 0 + 1), etc. Notice that in the *return* portion of the code that each *n* is **added** to the next one in the sequence of calls to *nrd*. Finally, when the *n* parameter coming into *adder* gets above 6, the returned value is n – 3  (obtained with 7 – 3 = 4). Thus, we have:

**0 + 1 + 2 + 3 + 4 + 5 + 6 + 4 = 25**

4. System.out.println(festus(0));  // **12**

```
public static int festus(int n)
{
        if (n > 6)
                return n - 3;
        else
        {
                n = n * 2;
                return n + festus(n + 1);
        }
}
```

On the first call to *festus*, *n* is 0 (and is modified to 0*2 = 0), and on the second call it's 1 (0 + 1 = 1, but quickly modified to 1 * 2 = 2), etc. Notice that in the *return* portion of the code that each **modified** *n* is **added** to the next one in the sequence of calls to *festus*. Finally, when the *n* parameter coming into *festus* gets above 6, the returned value is n – 3  (7 – 3 = 4). Thus, we have:

**0 + 2 + 6 + 4 = 12**

5. What is displayed by *homer(9);* ?  **1,2,4,9**

```
public static void homer(int n)
{
        if (n <= 1)
                System.out.print(n);
        else
        {
                homer(n / 2);
                System.out.print(“,” + n);
        }
}
```

Notice on this method that we successively pass in these values of *n*.

```
9    4    2    1
```

Nothing is printed until the last time when we are down to a 1. Then we start coming back up the calling chain and printing.

6. What is displayed by *method1(7);* ? **1,3,5,7**

```
public static void method1(int n)
{
        if (n <= 1)
                System.out.print(n);
        else
        {
                method1(n-2);
                System.out.print(“,” + n);
        }
}
```

7. In this problem we will generate the Fibonacci sequence. This important sequence is found in nature and begins as follows:

    0, 1, 1, 2, 3, 5, 8, 13, 21, …

    We notice that beginning with the third term, each term is the sum of the preceding two. Recursively, we can define the sequence as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n -2)
```

Using these three lines, we can write a recursive method to find the kth term of this sequence with the call, *System.out.println( fib(k) );* :

```
public static int fib(int n)
{
        if (n = = 0)
        {
                return 0;
        }
        else if(n = = 1)
        {
                return 1;
        }
        else
        {
                return fib(n – 1) + fib(n – 2);
        }
}
```

8. Let's try one similar to #7. What is returned by *pls(4);* ? **85**

```
public static int pls(int n)
{
        if (n = = 0)
        {
                return 5;
        }
        else if (n = = 1)
        {
                return 11;
        }
        else
        {
                return pls(n - 1) + 2 * pls(n - 2);
        }
}
```

The way we approach this is to just build the sequence from the rules we see expressed in the code. Term 0 has a value of 5 and term 1 has a value of 11.

| Term number | →0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Value | →5 | 11 | | | |

How will we get term 2? Well, the rule in the code says it's the previous term plus twice the term before that. That gives us 11 + 2*5 = 21. Continue this to obtain the other terms.

| Term number | →0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Value | →5 | 11 | 21 | 43 | **85** |

9. We are going to use these same ideas to easily work the next problem that in the beginning just looks hopelessly complicated.

```
public void f(int z)
{
        if(z = = 0)
        {
                System.out.print("x");
        }
        else
        {
                System.out.print("{");
                f(z-1);
                System.out.print("}");
        }
}
```

Let's begin analyzing this by observing the output of *f(0)*. It simply prints an "x".

| Term number | →0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Value | →x | | | |

Now, what about *f(1)*? It first prints a "{" followed by *f(z-1)*. But *f(z-1)* is simply the previous term, and we already know that it's an "x". A "}" follows. So our 2nd term is "{x}".

| Term number | →0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Value | →x | {x} | | |

Similarly, each subsequent term is the previous term sandwiched in between "{" and "}" and so we have:

| Term number | →0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Value | →x | {x} | {{x}} | {{{x}}} |

So, if we are asked for *f(3)* the answer is **{{{x}}}.**

10. What is returned by *g(6, 2)*?

```
public static void g(int x, int y)
{
      if (x/y != 0)
      {
         g(x/y, y);
      }
      System.out.print(x / y + 1);
}
```

To analyze this problem the following pairs will represent the parameters on subsequent recursive calls to *g*. Under each pair is what's printed.

```
6, 2            3, 2            1, 2
4               2               1
```

Realizing that we don't print until we reach the end of the calling chain, we see that **124** is printed as we "back-out" of the chain.

## Exercises on Lesson 40

In each of the following recursion problems, state what's printed.

1. System.out.println( rig(4) );

```
public static int rig(int n)
{
        if ( (n = = 0) )
        {
                return 5;
        }
        else if ( n = = 1)
        {
                return 8;
        }
        else
        {
                return rig(n – 1) - rig(n – 2);
        }
}
```

2. System.out.println( mm(6) );

```
public static int mm(int n)
{
        if (n<=0)
                return 10;
        else
                return n + mm(n-1);
}
```

3. System.out.println( adrml(5) );

```
public static int adrml(int n)
{
        if (n<=1)
                return n;
        else
                return n * adrml(n-2);
}
```

4. System.out.println(bud(1));

```
public static int bud(int n)
{
        if (n>5)
                return n - 2;
        else
                return n + bud(n +1);
}
```

5. System.out.println(zing(0));

```
public static int zing(int n)
{
        if (n > 10)
                return n - 2;
        else
        {
                n = n * 3;
                return n + zing(n + 2);
        }
}
```

6. crch(12);

```
public static void crch(int n)
{
        if (n <= 0)
                System.out.print(n);
        else
        {
                crch(n / 3);
                System.out.print(“,” + n);
        }
}
```

7. elvis(11);

```
public static void elvis(int n)
{
        if (n <= 3)
                System.out.print(n + 1);
        else
        {
                elvis(n-3);
                System.out.print(“>>” + (n – 1));
        }
}
```

8. sal(5);

```
public static int sal(int n)
{
        if (n = = 2)
        {       return 100;     }
        else if (n = = 3)
        {       return 200;     }
        else
        {
                return  (2 * sal(n - 1) +  sal(n - 2) + 1);
        }
}
```

9. puf(4);

```
public static void puf(int n)
{
        if(n = = 1)
        {       System.out.print(“x”);      }
        else if( n%2 = = 0) //n is even
        {
                System.out.print(“{”);
                puf(n-1);
                System.out.print(“}”);
        }
        else  //n is odd
        {
                System.out.print(“<”);
                puf(n-1);
                System.out.print(“>”);
        }
}
```

10. bc(6, 2);

```
public static void bc(int p, int q)
{
         if (p/q = = 0)
         {
                System.out.println(p + q + 1);
         }
        else
        {
                System.out.println(p);
                 bc(p/q, q);
        }
}
```

## Project… Fibonacci

You are to write a recursion routine to generate the kth term of a “modified” Fibonacci sequence. Our modified sequence will be defined as follows:

```
modFibonacci(0) = 3
modFibonacci(1) = 5
modFibonacci(2) = 8
modFibonacci(n) = modFibonacci(n - 1) + modFibonacci(n -2) + modFibonacci(n-3)
```

For your convenience several terms of this sequence are:

```
3     5     8     16    29    53    98 . . .
```

Call your new class *ModFib* and create a *static* method in it called *modFibonacci*.

Test your new class with the following *Tester* class:

```
import java.io.*;
import java.util.*;
public class Tester
{
   public static void main(String args[])
   {
      Scanner kbReader = new Scanner(System.in);
      System.out.print("Generate which term number? ");
      int k = kbReader.nextInt( );

      System.out.println("Term #" + k + " is " + ModFib.modFibonacci(k));
   }
}
```

Typical runs should look like this:

```
Generate which term number? 5
Term #5 is 53

Generate which term number? 6
Term #6 is 98
```

# Lesson 51….. Binary Search Tree

We will begin by showing how a binary search tree is constructed. We will construct our tree from the following sequence of integers; 50, 56, 52, 25, 74, and 54. Each number will result in a "node" being constructed. The nodes in the series of figures below are depicted with circles, and the sequence of figures shows the sequence in which the nodes are added.

**The rules:**

As can be observed from the drawings, a new integer (*n*) is added by starting at the root node (top level node) as the first "comparison node" and then using the following rules:

1. If the number we are to insert (*n*) is greater than the "comparison node", move down the tree and to the right; the first node encountered is the new "comparison node".
2. If the number we are to insert (*n*) is less than or equal to the "comparison node", move down the tree and to the left; the first node encountered is the new "comparison node".
3. If, after comparing at a node position, a node does not exist in the direction in which we try to move, then insert a new node containing the integer *n* at that position.

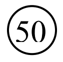


Fig 52-1 Tree started with root node

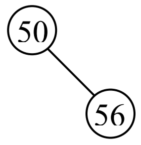


Fig 52-2 56 is added

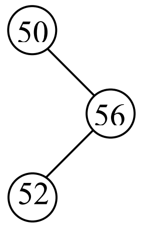


Fig 52-3 52 is added

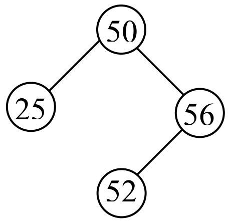


Fig 52-4 25 is added

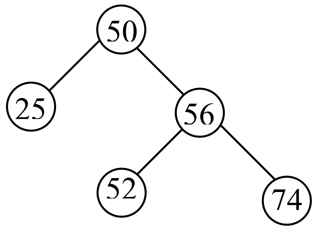


Fig 52-5 74 is added

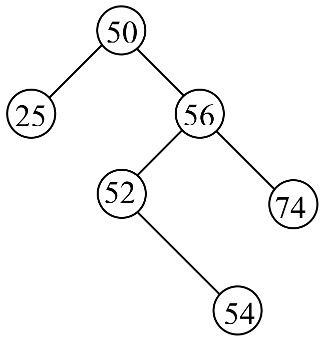


Fig 52-6 54 is added

**Creation of the *BSTNode* class:**

We will now create a class call *BST* (binary search tree) that will allow us to add nodes in the fashion described above. However, we must first have a class that creates the nodes themselves. What information must a node contain? For storing integers in the nodes, as we are doing in this example, each node should contain the following:

1. The actual data (an integer for the example above)
2. A reference to the right-hand node immediately beneath this node (*null* if it doesn't exist)
3. A reference to the left-hand node immediately beneath this node (*null* if it doesn't exist)

We are going to call this node-creating class, *BstNode*. Its implementation is shown below.

```
public class BstNode
{
        public BstNode(int i)  //Constructor
        {
                leftNode = null;
                rightNode = null;
                intData = i;
        }

        public int intData;
        public BstNode leftNode;
        public BstNode rightNode;
}
```

Notice that the three state variables in this class correspond to the three numbered requirements mentioned earlier. Also, notice that the *leftNode* and *rightNode* fields are set to *null* since when a node is constructed, there are no other nodes "hanging off it" yet.

**The *BST* class:**

Next, we turn our attention to the *BST* class itself. In the constructor, we simply create the root node (the highest level node).

The reader is strongly urged to look over Appendix W (Tree Definitions). There, you will get a firm grounding in the tree-terms we have already used here and new ones we will be using soon.

The constructor and state variables are as follows for the *BST* class:

```
public class BST
{
        public BST(int i) //constructor
        {    // Root node is instantiated at the time of creation of the tree object.
             rootNode = new BstNode(i);   //create a node with the above class
        }
         …more code to come…

        BstNode rootNode;
}
```

**The *addNode* method:**

Now comes the most important (and most complex) method of the *BST* class, the *addNode* method in which decisions are made as to the correct position for each new node to be added. Here are the rules for inserting a new node after first setting the root

1. If the number we are to insert (*n*) is greater than the *currentNode*, move down the tree and to the right; the first node encountered is the new *currentNode.*
2. If the number we are to insert (*n*) is less than or equal to the *currentNode*, move down the tree and to the left; the first node encountered is the new *currentNode.*
3. Continuing in this same fashion, move down the tree. If, after comparing at a node position, a node does not exist in the direction in which we try to move, then insert a new node containing the integer *n* at that position.

If these rules seem familiar, they are essentially the same as those at the top of page 52-1. Here, in this latest rendition, we become more specific with regard to the variable names to be used in the method that implements the rules. The complete class (including the *addNode* method) now reads:

```
public class BST
{
   public BST(int i)  //constructor: Root node added at the time of creation of the tree
   {
         rootNode = new BstNode(i);
   }

   public void addNode(int i)
   {
      BstNode currentNode = rootNode;
      boolean finished = false;
      do
      {
         BstNode curLeftNode = currentNode.leftNode;
         BstNode curRightNode = currentNode.rightNode;
         int curIntData = currentNode.intData;

         if(i > curIntData)
         {  //look down the right branch
                     if(curRightNode == null)
                     {  //create a new node  referenced with currentNode.rightNode
                           currentNode.rightNode = new BstNode(i);
                            finished = true;
                     }
                     else //keep looking by assigning a new current node one level down
                     {   currentNode = curRightNode;  }
         }
         else
         {  //look down the left branch
                     if(curLeftNode == null)
                     {  //create a new node referenced with currentNode.leftNode
                           currentNode.leftNode = new BstNode(i);
                           finished = true;
                     }
                     else
                     {  //keep looking by assigning a new current node one level down
                           currentNode = curLeftNode;
```

```
                }
            }
        }while(!finished);
    }
    BstNode rootNode;
}
```

It is left to the reader to examine the code in the *addNode* method and to convince himself that the three numbered rules are being implemented.

**A class for testing:**

To test the *BST* class, use the following *Tester* class:

```
public class Tester
{
        public static void main(String args[])
        {
                //the first integer in the tree is used to create the object
                BST bstObj = new BST(50);
                bstObj.addNode(56);
                bstObj.addNode(52);
                bstObj.addNode(25);
                bstObj.addNode(74);
                bstObj.addNode(54);
        }
}
```

**Prove that it really works:**

The integers mentioned in the example at the beginning of this lesson are added to the tree with this test code. But how do we really know that it is working? What we need is some type of printout. If we add the following *traverseAndPrint* method to the *BST* class we will see that our class does, indeed, perform as advertised.

```
public void traverseAndPrint(BstNode currentNode )
{
        System.out.print("data = " + currentNode.intData);
                //To aid in your understanding, you may want to just ignore this
                //indented portion and just print the integer. In that case, change the
                //line above to a println instead of a print.
                if(currentNode.leftNode == null)
                {     System.out.print("    left = null");     }
                else
                {     System.out.print("    left = " + (currentNode.leftNode).intData);   }

                if(currentNode.rightNode == null)
                {     System.out.print("    right = null");   }
                else
                {     System.out.print("    right = " + (currentNode.rightNode).intData);}
                System.out.println("");

        if(currentNode.leftNode != null)
            traverseAndPrint(currentNode.leftNode);
        if(currentNode.rightNode != null)
            traverseAndPrint(currentNode.rightNode);
}
```

This method has recursive calls to itself and will print every node in the tree. In addition to the data stored in each node (an integer), it also prints the contents of the two nodes "hanging off" this node.

Test this new method by adding the following code at the bottom of the *main* method in the *Tester* class.

```
//print all the nodes
bstObj.traverseAndPrint(bstObj.rootNode);
```

For the data suggested in the examples on page 52-1 the output will appear as shown below when the *main* method in the *Tester* class is executed:

```
data = 50    left = 25    right = 56
data = 25    left = null  right = null
data = 56    left = 52    right = 74
data = 52    left = null  right = 54
data = 54    left = null  right = null
data = 74    left = null  right = null
```

*************************************************************************

## Project… *BST find* Method

Now we come to what a binary search tree is all about, the search. You are to create a method of the *BST* class called *find*. Its signature is as follows:

```
public boolean find(int i)
```

It returns a *true* if *i* is found in the binary tree and *false* if it's not found. This method will use essentially the same rules as those for the *addNode* method except when we come to the place where we formerly added a node; we will exit the method and say that the search was unsuccessful. Likewise, there is more to the comparisons. We can no longer just test to see if the data we are searching for is greater than or less than that of the *currentNode*. We must now also test for equality.

To test the *find* method, add the following code to the bottom of the *main* method in *Tester*.

```
System.out.println(bstObj.find(74));  //This is one it will find…prints a true
System.out.println(bstObj.find(13));  //This is one it won't find…prints a false
```

**Why use a Binary Search Tree?**

What can searching a Binary Search Tree (BST) do that we could not accomplish searching a linear array? The BST can do it faster, much faster. The Big O value for a reasonably balanced BST is O(log n). For an unordered array it's O(n); however, for an ordered array, a binary search is also of the order O(log n). So, what are the advantages of a binary search tree over searching an ordered array (using a binary search) since their Big O values are the same? The advantages are:

1. Using a binary search on an array, **ordering is necessary** after the insertion of **each new element**. An alternative to this is **inserting** the new element in the correct position. In either case, the time required to do this is typically considerably more than the time required to insert a new node in a *BST*.

2. In an array, we must pre-dimension the array.
   a. If we dimension too small, we run the risk of running out of space if more nodes need to be added than were originally anticipated.
   b. If we dimension to large, we waste memory and may degrade the performance of the computer.

   With the *BST* object, we dynamically create nodes as we need them in dynamic memory. There is no need to know ahead of time how many there will eventually be.

**Anonymous objects:**

Have you noticed that with the *BST* class, the **node objects that contain our data are not named** (except for the root node)? We have to traverse the tree and each node we encounter gives references to the two nodes "hanging off" it with *leftNode* and *rightNode*. Recall that we had a similar situation in Lesson 49 with the singly linked list in which we had a "chain" of nodes, each with a reference to the next. Here, each node has references to **two** nodes that follow it.

**Balanced Tree:**

Above it was mentioned that the Big O value for searching a Binary Search Tree was O(log n) if the tree was reasonably balanced. What do we mean by a balanced tree? Refer back to Fig 52-6 and it can be seen that this tree is not balanced. There are more nodes to the right of the root (50) than to the left. An extreme case of this is shown below.

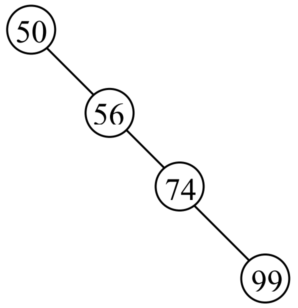


Fig 52-7 A totally unbalanced "tree"

Consider the following sequence of numbers to be added to a binary tree.

{50, 56, 74, 99}

The resulting "tree" to the left is totally unbalanced. Every new node to be added lies to the right of the previous node. In this case (which is clearly the worst case) the Big O value for searching the tree is O(n). If there are *n* items we might very well have to do *n* comparisons before finding the desired one.

If we are **very** unlucky, just such a tree **might** result when we add our nodes. With random data, it is not very likely to be as bad as Fig 52-7; however, what is more likely, is that that tree be somewhat out of balance which would, of course, reduce the efficiency of the search. What can we do to prevent this unbalance? It is beyond the scope of this book, however, there are algorithms that can detect this and "rebalance the tree". Nothing comes free, and this rebalancing adds complexity to the code as well as additional processing time.

**Generalizing, Using Objects for Data:**

It is possible to modify our class so that instead of just storing primitive integers we could store objects. To do this we would replace the code everywhere we pass an *int* variable as an argument, with *Comparable obj* .

The only catch is that the *obj* object that we pass in, **must implement the *compareTo* method**. The other requirement is that the former state variable, *int intData* be replaced with *Comparable data.* Rather than modify the *BST* class that we have already done, we are going to present another class that adds *Comparable* type objects to nodes in a Binary Search Tree. This class is just about the ultimate as far as brevity of code is concerned; however it is more difficult to understand because it uses recursion.

```
public class BsTree
{
   public BsTree(Comparable d)
   {
            theData = d;
            leftNode = null;     //This and next line could be omitted,
            rightNode = null;    //they are automatically null.
   }

   public BsTree addNode(Comparable d)
   {
            if(d.compareTo(theData) > 0)
            {    //d should be inserted somewhere in the branch to the right
                 if(rightNode != null)
                      //right node exists, go down that branch, look for place to put it
                       rightNode.addNode(d);
                 else
                      rightNode = new BsTree(d);   //Create new rightNode, store d in it
            }
            else
            {    //d should be inserted somewhere in the branch to the left
                 if(leftNode != null)
                      //left node exists, go down that branch, look for place to put it
                      leftNode.addNode(d);
                 else
                      leftNode = new BsTree(d);   //Create a new leftNode, store d in it
            }
            return this;
   }
   private Comparable theData;
   private BsTree leftNode, rightNode;
}
```

It is left to the reader to create a *find* method comparable to those of the *BST* class earlier in this lesson. We also need a *traverseAndPrint* method for this class. Three different versions of *traverseAndPrint* will be offered below as the various types of traversals are discussed.

**Traversal types:**
There are four traversal types. They are preorder, inorder, postorder, and level order traversals. Each visits **all** of the nodes in the tree, but each in a different order.

**Preorder traversal of a Binary Search Tree:**
Order of visitation of nodes: **50, 25, 18, 7, 19, 35, 30, 37, 76, 61, 56, 68, 80, 78, 85**

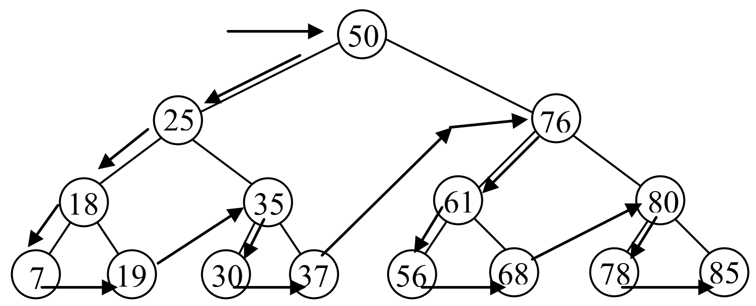


Fig. 52-8
Preorder traversal follows the sequence of arrows. **Rule: A node is visited before its descendants.**

The following code implements a preorder traversal of a tree as depicted in Fig. 52-8. An easy way to remember this code is to note the printing for this preorder traversal comes before the two recursive calls.

```
public void traverseAndPrint( ) //Use with BsTree class on previous page.
{
        System.out.println(theData);
        if( leftNode != null )  leftNode.traverseAndPrint( );
        if( rightNode != null )  rightNode.traverseAndPrint( );
}
```

**Inorder traversal of a Binary Search Tree:**
Order of visitation of nodes: **7, 18, 19, 25, 30, 35, 37, 50, 56, 61, 68, 76, 78, 80, 85**

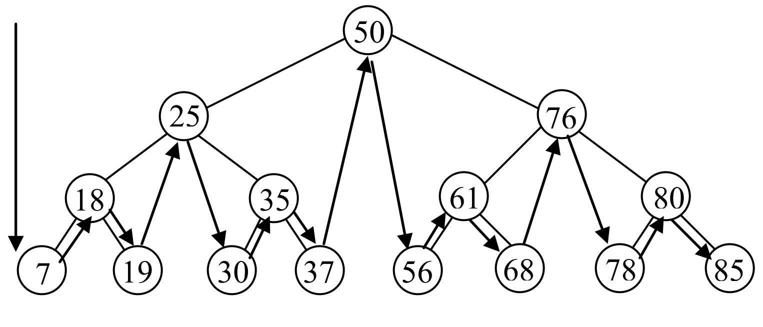


Fig. 52-9
Inorder traversal follows the sequence of arrows. The order is the ascending order of a sorted list. **Rule: A node is visited in-between its left subtree and right subtree. (Left visited first.)**

The following code implements an inorder traversal of a tree as depicted in Fig. 52-9. This technique is important since it **visits the nodes in a "sorted order."** An easy way to remember this code is to note the printing for this inorder traversal comes in-between the two recursive calls.

```
public void traverseAndPrint( )
```

```
        if( leftNode != null )  leftNode.traverseAndPrint( );
        System.out.println(theData);
        if( rightNode != null )  rightNode.traverseAndPrint( );
}
//Exchanging the first and last lines of this method results in a reverse-order traversal.
```

**Postorder traversal of a Binary Search Tree:**
Order of visitation of nodes: **7, 19, 18, 30, 37, 35, 25, 56, 68, 61, 78, 85, 80, 76, 50**

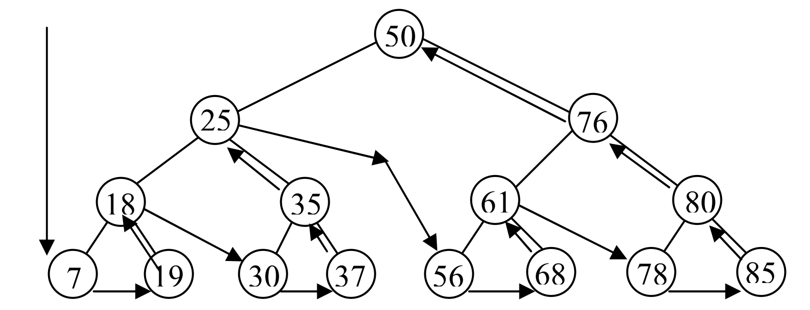


Fig. 52-10
Postorder traversal follows the sequence of arrows. **Rule: A node is visited after its descendants.**

The following code implements a postorder traversal of a tree as depicted in Fig. 52-10. An easy way to remember this code is to note the printing for this postorder traversal comes after the two recursive calls.

```
public void traverseAndPrint( )
{
        if( leftNode != null )  leftNode.traverseAndPrint( );
        if( rightNode != null )  rightNode.traverseAndPrint( );
        System.out.println(theData);
}
```

**Level order traversal of a Binary Search Tree:**
Order of visitation of nodes: **50, 25, 76, 18, 35, 61, 80, 7, 19, 30, 37, 56, 68, 78, 85**

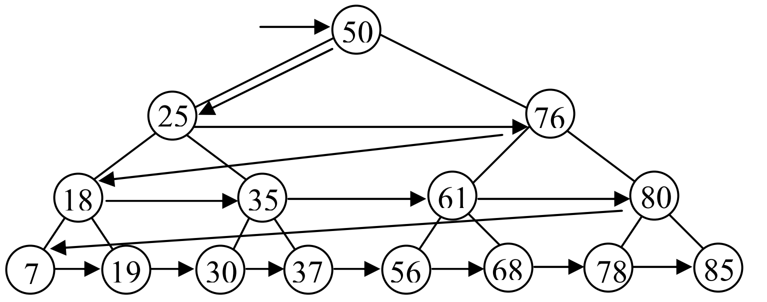


Fig. 52-11
Level order traversal follows the sequence of arrows.

The code that would implement this is a bit more involved than the others. One way to do it is to have counters that keep up with how deep we are in the tree.

## An Application of Binary Trees… Binary Expression Trees

Consider the infix expressions (6 + 8) * 2 and 5 + (3 * 4). The expression trees to the right are a result of parsing these expressions. As can be inferred from the drawings, the following rules apply for an expression tree:

- Each leaf node contains a single operand.
- Each interior node contains an operator.
- The left and right subtrees of an operator node represent subexpressions that must be evaluated **before** applying the operator at the operator node.
    - The levels of the nodes in the tree indicate their relative precedence of evaluation.
    - Operations at the lower levels must be done **before** those above them.
    - The operation at the root of the tree will be the last to be done.

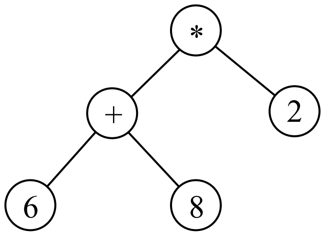


Fig. 52-12 (6 + 8) * 2

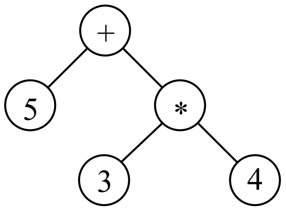


Fig 52-13 5 + (3 * 4)

We will now look at a larger expression tree and see how the inorder, preorder, and postorder traversals of the tree have special meanings with regard to the mathematics of an expression.

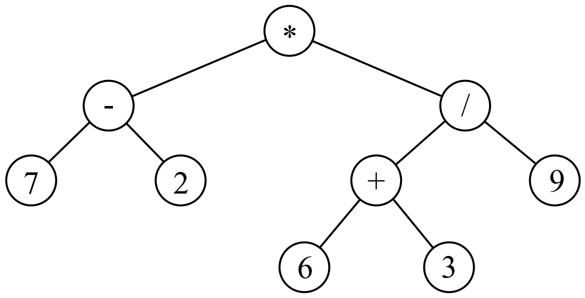


Fig. 52-14 A binary expression tree for the infix expression (7 - 2) * ( (6+3) / 9)

An **Inorder Traversal** of the above expression tree yields the **infix** form: (7 - 2) * ( (6+3) / 9)

A **Preorder Traversal** of the above expression tree yields the **prefix** form: * - 7 2 / + 6 3 9

A **Postorder Traversal** of the above expression tree yields the **postfix** form: 7 2 - 6 3 + 9 / *
Notice that the postfix form is Reverse Polish Notation (RPN), the form that was used for the stack calculator of Lesson 50.

## Binary Search Tree… Contest Type Problems

1. Which of the following replaces **<*1>** in the code to the right to make the *traverseAndPrint* method visit and print every node in a "Postorder" fashion?

    A. 
    ```
    if(leftNd != null)  leftNd.traverseAndPrint( );
    System.out.print(info);
    if(rightNd!=null) rightNd.traverseAndPrint( );
    ```

    B. 
    ```
    if(leftNd != null)  leftNd.traverseAndPrint( );
    if(rightNd!=null) rightNd.traverseAndPrint( );
    System.out.print(info);
    ```

    C. 
    ```
    System.out.print(info);
    if(leftNd != null)  leftNd.traverseAndPrint( );
    if(rightNd!=null) rightNd.traverseAndPrint( );
    ```

    D. 
    ```
    leftNd.traverseAndPrint( );
    rightNd.traverseandPrint( );
    ```

    E. None of these

2. Assume **<*1>** has been filled in correctly. Which of the following creates a *Bst* object *obj* and adds 55 as a wrapper class Integer?

    A. 
    ```
    Integer J;
    J = 55;
    Bst obj = Bst(J);
    ```

    B. `Bst obj = new Bst( new Integer(55) );`

    C. 
    ```
    Bst obj;
    obj.addNd(55);
    ```

    D. 
    ```
    Bst obj;
    obj.addNd( new Integer(55) );
    ```

    E. None of these

3. Assume **<*1>** has been filled in correctly and that *n* objects are added to an object of type *Bst* in order from largest to smallest. What is the Big O value for searching this tree?

    A. O(n log n)
    B. O(log n)
    C. O(n)
    D. O($n^2$)
    E. None of these

```
//Binary Search Tree
public class Bst
{
    public Bst(Comparable addValue)
    {
        info = addValue;
    }

    public Bst addNd(Comparable addValue)
    {
        int cmp = info.compareTo(addValue);

        if(cmp<0)
        {
            if(rightNd!=null)
                rightNd.addNd(addValue);
            else
                rightNd=new Bst(addValue);
        }
        else if(cmp>0)
        {
            if(leftNd!=null)
                leftNd.addNd(addValue);
            else
                leftNd=new Bst(addValue);
        }
        return this;
    }

    public void traverseAndPrint( )
    {
        <*1>
    }

    private Comparable info;
    private Bst leftNd;
    private Bst rightNd;
}
```

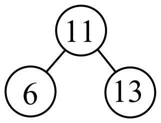
4. When a *Bst* object is constructed, to what value will *leftNd* and *rightNd* be initialized?

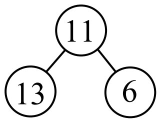
A. this
B. 0
C. null
D. Bst object
E. None of these

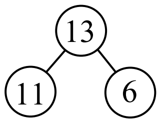
5. After executing the code below, what does the resulting tree look like?

```
Bst obj = new Bst(new Integer(11));
obj.add(new Integer(6))
obj.add(new Integer(13));
```

A. ArithmeticException

B.


C.


D.


E. None of these

6. What replaces **<*1>** in the code to the right so that a "Preorder" traversal is done?

A.
```
if(leftNd != null)  leftNd.traverseAndPrnt( );
System.out.print(info);
if(rightNd!=null)rightNd.traverseAndPrnt();
```

B.
```
if(leftNd != null)  leftNd.traverseAndPrnt( );
if(rightNd!=null)rightNd.traverseAndPrnt();
System.out.print(info);
```

C.
```
System.out.print(info);
if(leftNd != null)  leftNd.traverseAndPrnt( );
if(rightNd!=null)rightNd.traverseAndPrnt();
```

D.
```
leftNd.traverseAndPrnt( );
rightNd.traverseandPrnt( );
```

E. None of these

```
//Binary Search Tree
public class Bst
{
    public Bst(Comparable addValue)
    {
        info = addValue;
    }

    public Bst addNd(Comparable addValue)
    {
        int cmp = info.compareTo(addValue);

        if(cmp<0)
        {
            if(rightNd!=null)
                rightNd.addNd(addValue);
            else
                rightNd=new Bst(addValue);
        }
        else if(cmp>0)
        {
            if(leftNd!=null)
                leftNd.addNd(addValue);
            else
                leftNd=new Bst(addValue);
        }
        return this;
    }

    public void transverseAndPrnt( )
    {
        <*1>
    }

    private Comparable info;
    private Bst leftNd;
    private Bst rightNd;
}
```

7. What is a disadvantage of an unbalanced Binary Search Tree?

A. No disadvantage B. Uses excessive memory C. Limited accuracy
D. Reduced search efficiency E. None of these

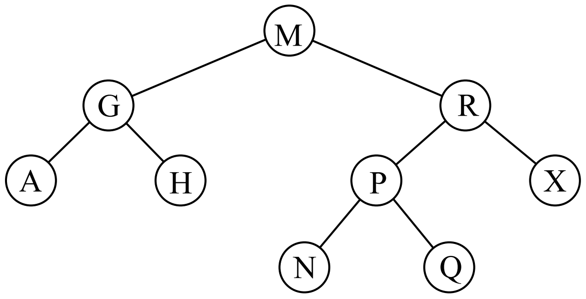
8. Average case search time for a Binary Search Tree that is reasonably balanced is of what order?

A. O(n log n) B. $O(n^2)$ C. O(n) D. O(1) E. None of these

9. What positive thing(s) can be said about a completely unbalanced tree that results from adding the following integers to a tree in the sequence shown?

{ 5, 6, 7, … 999, 1000}

A. The items are automatically in numerical order along the long sequential strand.
B. The smallest number is automatically the root node. C. The largest number is the root node.
D. Both A and B E. Both A and C



10. In what order are the nodes visited in the tree to the left if a preorder traversal is done?

A. A, G, H, M, N, P, Q, R, X
B. M, G, A, H, R, P, N, Q, X
C. A, H, G, N, Q, P, X, R, M
D. M, G, R, A, D, P, X, N, Q
E. None of these

11. In what order are the nodes visited in the tree to the left if a postorder traversal is done?

A. A, G, H, M, N, P, Q, R, X
B. M, G, A, H, R, P, N, Q, X
C. A, H, G, N, Q, P, X, R, M
D. M, G, R, A, H, P, X, N, Q
E. None of these

12. In what order are the nodes visited in the tree to the left if an inorder traversal is done?

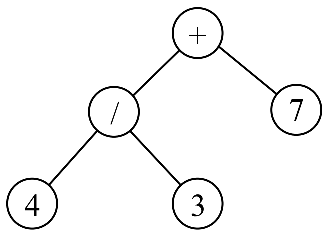
A. A, G, H, M, N, P, Q, R, X
B. M, G, A, H, R, P, N, Q, X
C. A, H, G, N, Q, P, X, R, M
D. M, G, R, A, H, P, X, N, Q
E. None of these

13. For the tree above, which of the following is a possible order in which the nodes were originally added to the binary search tree?

A. M, G, R, A, H, X, P, N, Q B. M, G, R, A, H, Q, N, P, X
C. M, R, A, G, H, X, P, N, Q D. A, G, H, M, N, P, Q, R, X
E. None of these

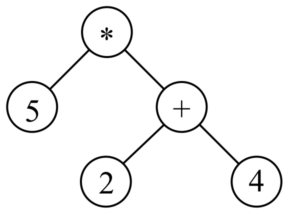
14. What mathematical infix expression is represented by the binary expression tree to the right?

A. (4 + 3) / 7
B. 4 / (3 + 7)
C. 7 / 4 / 3 + 7
D. (4 / 3) + 7
E. None of these



15. What mathematical infix expression is represented by the binary expression tree to the right?

A. 5 * 2 + 4
B. 5 * (2 + 4)
C. (2 * 4) + 5
D. 5 * 2 * (+4)
E. None of these



16. Which of the following is a postfix version of the following mathematical expression?

(37 - 59) * ( (4 + 1) / 6 )

A. * - 37 59 / + 4 1 6
B. (37 - 59) * ( (4 + 1) / 6 )
C. 37 59 - 4 1 + 6 / *
D. 37 - 59 * 4 + 1 / 6
E. None of these

17. What is the minimum number of levels for a binary tree with 20 nodes?

A. 20 B. 7 C. 6 D. 5 E. None of these

18. What is the maximum number of levels for a binary tree with 20 nodes?

A. 20 B. 7 C. 6 D. 5 E. None of these

# Golden Nugget of Wisdom # 20

**1. Initialization blocks** are blocks of code embedded within a class, and as the name implies, they are mostly used to initialize variables. **Multiple** initialization blocks are possible as is shown in the sample class below:

```
public class DemoClass
{
        //Non-static initialization block
        {   stateVar1 = 50;   }

        //Static initialization block
        static  //To manipulate static variables, use a static initialization block
        {   stateVar2 = 20;   }

        public DemoClass( )  //constructor
        {
                stateVar1++;
                stateVar2--;
        }

        … Methods and other state variables…

        public int stateVar1;  //If initialization blocks exist above don't do any
        public static int stateVar2;                        //initializing here.
}
```

**2. Rules for initialization blocks:**

- Non-static blocks run every time an object is created.
- Static blocks run just once (when the class is first loaded).
- Blocks are executed in the order in which they occur.
- Regardless of placement, code in the blocks executes **before** constructor code.

**3. Sample usage:**

```
DemoClass demo1 = new DemoClass( );
System.out.println( demo1.stateVar1 + "    " + demo1.stateVar2);  //51   19
DemoClass demo2 = new DemoClass( );
System.out.println( demo2.stateVar1 + "    " + demo2.stateVar2);  //51   18
```

Initialization blocks are rarely used and there really is no point in using them as in the two sample blocks above. It would be more straightforward to just initialize these two state variables on the bottom two lines where they are declared. So, what is the real purpose of initialization blocks? Suppose we have a program that absolutely must run as fast as possible; however, it has loops that require the laborious, time-consuming calculation of something like *Math.tan(Math.log(Math.sqrt(1- x* x)))* for values of *x* ranging from 1 to 360 in increments of 1. In this case it would be wise to iterate 360 times through a loop in an initialization block and precalculate all these values and store in a state variable array such as *double val[ ]*. Then in the actual program, when needed, quickly access the desired value with *val[x]*.

# Appendix G …..Two's Complement Notation

The two's complement notation is the protocol used to store **negative numbers**. Let's consider the integer (4 bytes) 13 in its binary form:

$00000000\ 00000000\ 00000000\ 00001101_{bin} = 13_{dec}$

What could we do to make this a negative number? The way we approach this is to think about negative 13 in this way:

13 + (negative 13) = 0

**So, our requirement will be that negative 13 be represented in such a way that when added to 13 it will give a result of 0**.

We will begin by adding the original binary form of 13 to the ones' complement (invert, 1's changed to 0's and vice versa) of 13.

```
 00000000  00000000  00000000  00001101
 11111111  11111111  11111111  11110010
 ______________________________________
 11111111  11111111  11111111  11111111
```

This is **not** what we want. We want all zeros; however, notice if we add 1 to this answer a carry will "ripple" all the way through, and if we just ignore the last carry on the end, we have our answer of 0.

```
 11111111  11111111  11111111  11111111
                                      1
 ______________________________________
100000000  00000000  00000000  00000000
|
Ignore this last carry
```

So, the way to get –13 is to invert 13 and add 1.

```
00000000  00000000  00000000  00001101  (13 in binary)

11111111  11111111  11111111  11110010  (13 inverted)
                                     1  (add 1)
______________________________________
11111111  11111111  11111111  11110011  (two's complement form of –13)
```

**Rules/Observations:**

1. To produce the negative of a number (two's complement form), perform the following three steps.
   a. Express the number in binary form
   b. Invert the number (change 1's to 0's and vice versa)
   c. Add 1
2. Negative numbers will always have a most significant bit (msb) value of 1.
3. Positive numbers will always have an msb value of 0.

4. This msb is known as the **sign bit** and does **not** have a positional value as do the other bits.

As an interesting exercise, you might try the following code.

```
int x = ???;  // enter any number you like for ???
System.out.println( x + (~x + 1) );  //prints 0 for any value of x
                                     //Notice you are inverting x and adding 1 to produce
                                     //the negative of x.
```

****************************************************************************

We are now going to take a completely different approach to ten's complement and see that when extending this idea to the binary system, we would have the two's complement.

Consider an old-fashioned car mileage indicator (odometer). If the register rotates forward, it performs addition one mile at a time. If the register rotates backward, it performs subtraction one mile at a time. Below is a five-digit register rotating backwards:

```
00004
00003
00002
00001
00000
99999
99998
99997
```

What we have done here is to work the problem 4 – 7, because we started with 4 and then rotated backwards 7 places. The answer is, of course, –3. However, the 99997 we got is what we call the ten's compliment of 3. In other words, 99997 is one way to represent –3. To see that 99997 really corresponds to –3, let's work the problem 4 + (-3) and see if we get +1.

```
 00004
 99997 (This corresponds to –3)
100001 (This is the answer if we ignore the "left-most" carry.)
```

Similarly, a backwards rotating "binary" odometer would look like this:

```
0100
0011
0010
0001
0000
1111
1110
1101
```

Again, what we are doing here is working the problem 4 – 7, because we start with 4 and rotate backwards 7 places. The answer is –3 and the $1101_{bin}$ we get is what we call the two's complement of 3. To see if this really works, let's do the problem 4 + (-3) and see if we get 1.

$$\begin{array}{r@{\;=\;}l} 0100_{\text{bin}} & 4_{\text{dec}} \\ \underline{1101}_{\text{bin}} & -3_{\text{dec}} \\ 10001_{\text{bin}} & 1 \text{ (ignoring the "left-most" carry)} \end{array}$$

Notice that the two's compliment representation of $-3_{dec}$ ($1101_{bin}$) is exactly what we would get from the previous discussion where we would have inverted and added 1.

# Appendix P ….. Time Allocation for Lessons and Tests

| Lesson | Time | Comments |
|---|---|---|
| "First day" activities | 1 day | Pass out books, demonstrate how to log-on, create project folder, learn how to launch and configure programming environment. |
| Lesson 1 | 1 day | Enter the "Hello World" program into the computer and execute. |
| Lesson 2 | 1 day | Illustrate each point of Lesson 2 by modifying the code of the "Hello World" program of Lesson 1. Assign the exercise on Lesson 2 as homework. There should be some time to work on this towards the end of the period and time to check answers. |
| Lesson 3 | 2 days | Illustrate each point of Lesson 3 by running code. Assign Exercise on Lesson 3. Grade assignment at end of 2nd day. |
| **Test through Lesson 3** | 1 day | Allow the students to work on the test, take it up at the end of the class, and let them know when them come back the next day they can make any corrections they look up that night. Keep beginning students from becoming discouraged from the start by making this an open-book test. |
| Lesson 4 | 2 days | Have the students run many of the code examples in their IDE (integrated development environment) Some of the problems on the exercise must be finished as homework in order to fit this lesson into one day. |
| Lesson 5 | 1 day | Run several of the code examples. |
| Lesson 6 | 1 day | Run several of the code examples. |
| Lesson 7 | 1 day | Do the first project together in class and assign the "Full Name" project as a written assignment. Typical grades range from 92 to 51. Problems 5, 9, 11, 12, 2, and 25 were those most often missed. On the day after the test go over these specific problems. |
| **Test through Lesson 7** | 1 day | At this point still let the students use the book for the test… try to build up their confidence. |
| Lesson 8 | 2 days | Many of the code example need to be run on the computer as they are discussed. |
| Lesson 9 | 2 days | Be sure to actually run the first two code examples. |
| Lesson 10 | 3 days | Definitely run the code "menu" example. |
| **Test through Lesson 10** | 1 day | This test may be difficult for some. On the day before the test let the students look over a copy of the test for about 10 minutes so there will be no surprises on the day of the test. |
| Lesson 11 | 3 days | This is the most important lesson so far. Be sure to run several of the code examples. This is where we begin to acclimate the students to "contest type problems". |
| Lesson 12 | 3 days | Again, run many of the code examples… very important concepts here. After the 14 regular exercise problems are completed, give the 5 "contest type" problems as a quiz. |
| Lesson 13 | 2 days | Emphasize the techniques for storing a char into a *String* and vice versa. Have students memorize the ranges of ASCII codes. |
| Lesson 14 | 2 days | Use chalk board for demo of conversion techniques. |
| **Test through Lesson 14** | 2 days | This will be a lengthy test and it is suggested that it not be an open-book test. Many students will need two days. Let the first day be an eye-opener for them so they will study overnight and continue the next day. |
| Make-up test through Lesson 14 (**Alternate Test through Lesson 14**) | 2 days | In order not to discourage students who do poorly on the original test, you might possibly want to give this 16-question re-take. Spend one day correcting the mistakes from the original and then one day taking this new test… The questions on this test are mostly what is likely to have been missed on the original test. |
| Lesson 15 | 3 days | Spend time on this lesson! This is the most important lesson so far. Have students enter and test the code for the *Circle* class. There are 20 questions on the exercise for this lesson. If the students do poorly on the exercise there is a "redemptive" quiz that could be given. |
| Lesson 16 | 3 days | This is a follow up to lesson 15…very important concepts here. |

| | | |
|---|---|---|
| **Test through Lesson 16** | 1 day | This is primarily a test on objects and classes (lesson 15 and 16). |
| Lesson 17 | 3 days | Students will find this much easier than the previous lessons on objects. |
| Lesson 18 | 3 days | Let students know that arrays will be used in nearly all future lessons. |
| Lesson 19 | 4 days | Be sure to do the programming projects. |
| **Test through Lesson 19** | 2 days | It is suggested that this test be split across two days. Overnight they can study what they saw on the test and didn't understand. |
| Lesson 20 | 2 days | |
| Lesson 21 | 1 day | |
| Lesson 22 | 1 day | |
| **Test through Lesson 22** | 2 days | This is a difficult test. Let the students work on it for 30 minutes or so the first day, take it up, let them study overnight, and then finish the second day. |
| Lesson 23 | 4 days | Plan to spend 1 day going over the material in the textbook. The second day can be devoted to doing and explaining the exercises. The programming project will also take a complete period if the students are forced to do most of it themselves. The contest type problems will require a day. Some of those problems are tricky and will require some explanation. |
| Lesson 24 | 3 days | Many lesson from this point on depend on inputting data from a file. Make sure the students get a good foundation. |
| **Test through Lesson 24** | 1 day | This test is considerably shorter than the others and probably easier. |
| Lesson 25 | 3 days | Honing skills with file input. |
| Lesson 26 | 1 day | Actually this lesson can be done in half a period. |
| Lesson 27 | 2 days | Be sure students keep the *BaseClass* class. They will paste code from it into many of their future projects. |
| **Test through Lesson 27** | 1 day | |
| Lesson 28 | 3 days | The project in this lesson will take an entire day for most students. |
| Lesson 29 | 2 days | |
| **Test through 29** | 1 day | |
| Lesson 30 | 3 days | Be sure to do the Monte Carlo project. |
| Lesson 31 | 1 day | Stress the *append* and *toString* methods. |
| Lesson 32 | 3 days | DeMorgan's theorem is very important. |
| Lesson 33 | 1 day | |
| **Test through 33** | 1 day | |
| Lesson 34 | 2 days | Some important concepts are here. |
| Lesson 35 | 3 days | |
| **Test through 35** | 1 day | |
| Lesson 36 | 3 days | This can be done in three days; however, this is such an important lesson that it might be more desirable to allocate 4 days. |
| **Test through 36** | 1 day | This test focuses strictly on Lesson 36, the inheritance lesson. |
| Lesson 37 | 2 days | |
| Lesson 38 | 2 days | |
| **Test through 38** | 1 day | |
| Lesson 39 | 2 days | |
| Lesson 40 | 3 days | This lesson on recursion is especially important. |
| **Test through 40** | 1 day | |
| Lesson 41 | 6 days | Spend one day for each sorting type. |
| **Test on Lesson 41** | 1 day | |
| Lesson 42 | 1 day | |
| Lesson 43 | 3 days | ArrayList. Spend at least one day on the project. |
| Lesson 44 | 4 days | Iterators |
| **Test on Lesson 44** | 1 day | |
| Lesson 45 | 3 days | These concepts are very important. Be sure to do all three projects. |
| **Test on Lesson 45** | 1 day | |
| Lesson 46 | 2 days | |
| Lesson 47 | 3 days | |
| **Test on Lesson 47** | 1 day | |
| Lesson 48 | 2 days | |
| Lesson 49 | 3 days | |

| | | |
|---|---|---|
| Lesson 50 | 2 days | |
| **Test on Lesson 50** | 1 day | |
| Lesson 51 | 3 days | |
| Lesson 52 | 3 days | |
| **Test on Lesson 52** | 1 day | |
| Lesson 53 | 2 days | |
| Lesson 54 | 2 days | |
| **Test on Lesson 54** | 1 day | |
| Lesson 55 | 3 days | |
| Lesson 56 | 3 days | |
| Lesson 57 | 3 days | |
| **Test on Lesson 57** | 1 day | |
| | | |

# Appendix Q ….. AP (A) Correlation

Not all of the following items are tested in the AP A test, but all have at least “potential relevance” as described in the AP Java subset on the College Board web site.

| Items on the A test | Page numbers |
|---|---|
| int, double | 2-1 |
| +, -,*, /, ++, --, % | 4-2 |
| = =, !=, >, <, >=, <= | 8-1, 9-1 |
| &&, \|\|, ! | 8-1 |
| Casting (int), (double) | 5-1 |
| String concatenation | 3-1 |
| Escape sequences \”, \\, \n | 3-1, C-1 |
| System.out.print( ) and System.out.println( ) | 1-1, 1-2 |
| One-dimensional arrays | 18-1—18-7 |
| Two-dimensional arrays | 35-1 |
| if, if/else | 9-1 |
| while, do/while | 12-1 |
| for | 11-1 |
| Design new and modify existing classes | 15-1—16-7 |
| return types | 15-1 |
| public classes, private instance variables, public and private methods | 15-1—16-7 |
| final local variables | 5-1 |
| final class, final methods | 36-2, 3 |
| static methods | 19-3, 20-1 |
| null | Nugs-17 |
| this | 36-3, 36-11__36-15, 46-6 |
| super | 36-1, 3,7, 36-12—36-15 |
| Constructors | 15-1 |
| static variables | 20-1 |
| static methods | 20-1 |
| Inheritance hierarchies | 36-1—36-15 |
| Modifying and creating subclasses | 36-1—36-15 |
| Modifying, creating, and implementing interfaces | 38-1—38-8 |
| abstract classes and abstract interfaces | 38-1 |
| equals method for objects | 9-1, 16-2 |
| = = and != for objects | 16-2 |
| Comparison of objects with Comparable.compareTo | 45-1 |
| Conversion to supertypes and subtype casts | 36-4, 45-3—45-4 |
| Package concepts, creating, importing | 7-1, 19-3, I-1, M-1 |
| Exceptions concepts; checked and unchecked | 37-1—37-11 |
| String | 2-1, 3-1 |
| Math class (abs, pow, sqrt, random) | 6-1 |
| Object | 36-4 |
| ArrayList | 43-1 |
| Wrapper Classes; Double, Integer | 21-1 |
| Sorting methods (not including Quick Sort) | 19-3, 41-1—41-17 |

| | |
|---|---|
| List interface (size, add, get, set, remove) | 42-1 |
| Binary Search | 51-1 |
| Enhanced for-loop | 19-5, Nug-16, 44-3 |
| Recursion | 40-1 |
| | |

The computer science **"case study", Grid World**, is also covered on the A test. This is thoroughly presented and explained in Blue Pelican's Grid World product in the form of documents, lessons, questions/answers, and videos… available at www.bluepelicanjava.com.

# Appendix R… Texas TEKS Correlation, Computer Science I

| Texas TEKS (Knowledge and Skills) | Student Expectations | Page(s) |
|---|---|---|
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | A. Demonstrate knowledge and appropriate use of operating systems, software applications, and communication and networking components. | S-4, U-1 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | B. Compare, contrast, and appropriately use the various input, processing, output, and primary/secondary storage devices. | S-5 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | C. Make decisions regarding the selection, acquisition, and use of software taking under consideration its quality, appropriateness, effectiveness, and efficiency. | 14-4, U-1 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | D. Delineate and make necessary adjustments regarding compatibility issues including, but not limited to, digital file formats and cross platform connectivity. | E-2, T-2 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | E. Differentiate current programming languages, discuss the use of the languages in other fields of study, and demonstrate knowledge of specific programming terminology and concepts. | V-1, V-2 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | F. Differentiate among the levels of programming languages including machine, assembly, high-level compiled and interpreted languages. | V-1, V-2 |
| 01. Foundations. The student demonstrates knowledge and appropriate use of hardware components, software programs, and their connections. | G. Demonstrate coding proficiency in a contemporary programming language. | Lessons 1 - 48 |
| 02. Foundations. The student uses data input skills appropriate to the task. | A. Demonstrate proficiency in the use of a variety of input devices such as keyboard, scanner, voice/sound recorder, mouse, touch screen, or digital video by appropriately incorporating such components into the product. | 7-1, 45-5, U-1 |
| 02. Foundations. The student uses data input skills appropriate to the task. | B. Use digital keyboarding standards for the input of data. | 1-1, 7-1 |
| 03. Foundations. The student complies with the laws and examines the issues regarding the use of technology in society. | A. Discuss copyright laws/issues and model ethical acquisition and use of digital information, citing sources using established methods. | T-2 |
| 03. Foundations. The student complies with the laws and examines the issues regarding the use of technology in society. | B. Demonstrate proper etiquette and knowledge of acceptable use policies when using networks, especially resources on the Internet and intranet. | T-2 |
| 03. Foundations. The student complies with the laws and examines the issues regarding the use of technology in society. | C. Investigate measures, such as passwords or virus detection/prevention, to protect computer systems and databases from unauthorized use and tampering. | 47-2, T-2 |
| 03. Foundations. The student complies with the laws and examines the issues regarding the use of technology in society. | D. Discuss the impact of computer programming on the World Wide Web (WWW) community. | 36-5, V-1 |
| 04. Information acquisition. The student uses a variety of strategies to acquire information from electronic resources, with appropriate supervision. | A. Use local area networks (LANs) and wide area networks (WANs), including the Internet and intranet, in research and resource sharing. | U-1 |
| 04. Information acquisition. The student uses a variety of strategies to acquire information from electronic resources, with appropriate supervision. | B. Construct appropriate electronic search strategies in the acquisition of information including keyword and Boolean search strategies. | 8-1, 8-3 |
| 05. Information acquisition. The student acquires electronic information in a variety of formats, with appropriate supervision. | A. Acquire information in and knowledge about electronic formats including text, audio, video, and graphics. | 14-4, E-1, E-2, E-3 |

| | | |
|---|---|---|
| 05. Information acquisition. The student acquires electronic information in a variety of formats, with appropriate supervision. | B. Use a variety of resources, including foundation and enrichment curricula, together with various productivity tools to gather authentic data as a basis for individual and group programming projects. | 14-4, U-1 |
| 05. Information acquisition. The student acquires electronic information in a variety of formats, with appropriate supervision. | C. Design and document sequential search algorithms for digital information storage and retrieval. | 39-3, 41-2, 47-1 |
| 06. Information acquisition. The student evaluates the acquired electronic information. | A. Determine and employ methods to evaluate the design and functionality of the process using effective coding, design, and test data. | 7-3, 11-5, 15-8, 16-6, 17-6, 23-5, 24-5 |
| 06. Information acquisition. The student evaluates the acquired electronic information. | B. Implement methods for the evaluation of the information using defined rubrics. | U-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | A. Apply problem-solving strategies such as design specifications, modular top-down design, step-wise refinement, or algorithm development. | 27-3, 27-4, L-1, 25-6, 30-6 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | B. Use visual organizers to design solutions such as flowcharts or schematic drawings. | 48-1, 48-2 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | C. Develop sequential and iterative algorithms and code programs in prevailing computer languages to solve practical problems modeled from school and community. | 25-6, 26-2, 27-4, 38-7 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | D. Code using various data types. | 2-1, 8-1, 10-1, 18-1, D-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | E. Demonstrate effective use of predefined input and output procedures for lists of computer instructions including procedures to protect from invalid input. | 37-1, 38-1, 42-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | F. Develop coding with correct and efficient use of expressions and assignment statements including the use of standard/user-defined functions, data structures, operators/proper operator precedence, and sequential/conditional/repetitive control structure. | 4-1, 6-1, 8-1, 9-1, 10-1, 12-1, H-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | G. Create and use libraries of generic modular code to be used for efficient programming. | 6-1, 19-3, 21-1, 23-1, 31-1, 37-1, 46-1, 47-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | H. Identify actual and formal parameters and use value and reference parameters. | 15-2, 15-3, 34-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | I. Use control structures such as conditional statements and iterated, pretest, and posttest loops. | 9-1, 10-1, 11-1, 12-1 |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | J. Use sequential, conditional, selection, and repetition execution control structures such as menu-driven programs that branch and allow user input. | 9-1, 7-1, 10-1, |
| 07. Solving problems. The student uses appropriate computer-based productivity tools to create and modify solutions to problems. | K. Identify and use structured data types of one-dimensional arrays, records, and text files. | 18-1, 19-1, 24-1, F-1 |
| 08. Solving problems. The student uses research skills and electronic communication, with appropriate supervision, to create new knowledge. | A. Participate with electronic communities as a learner, initiator, contributor, and teacher/mentor. | 36-5, U-1 |
| 08. Solving problems. The student uses research skills and electronic communication, with appropriate supervision, to create new knowledge. | B. Demonstrate proficiency in, appropriate use of, and navigation of LANs and WANs for research and for sharing of resources. | 47-2, T-2, U-1 |

| | | |
|---|---|---|
| 08. Solving problems. The student uses research skills and electronic communication, with appropriate supervision, to create new knowledge. | C. Extend the learning environment beyond the school walls with digital products created to increase teaching and learning in the foundation and enrichment curricula. | 14-4, U-1 |
| 08. Solving Problems. The student uses research skills and electronic communication, with appropriate supervision, to create new knowledge. | D. Participate in relevant, meaningful activities in the larger community and society to create electronic projects. | 36-5, U-1 |
| 09. Solving problems. The student uses technology applications to facilitate evaluation of work, both process and product. | A. Design and implement procedures to track trends, set timelines, and review/evaluate progress for continual improvement in process and product. | 39-1, 41-2, 41-4, 41-6, 41-9 |
| 09. Solving problems. The student uses technology applications to facilitate evaluation of work, both process and product. | B. Use correct programming style to enhance the readability and functionality of the code such as spacing, descriptive identifiers, comments, or documentation. | 1-2, 2-2, 15-1 |
| 09. Solving problems. The student uses technology applications to facilitate evaluation of work, both process and product. | C. Seek and respond to advice from peers and professionals in delineating technological tasks. | 36-5, U-1 |
| 09. Solving problems. The student uses technology applications to facilitate evaluation of work, both process and product. | D. Resolve information conflicts and validate information through accessing, researching, and comparing data. | 45-1, 45-5, U-1 |
| 09. Solving Problems. The student uses technology applications to facilitate evaluation of work, both process and product. | E. Create technology specifications for tasks/evaluation rubrics and demonstrate that products/product quality can be evaluated against established criteria. | 14-4, U-1 |
| 10. Communication. The student formats digital information for appropriate and effective communication. | A. Annotate coding properly with comments, indentation, and formatting. | 1-2, 2-2, 27-3 |
| 10. Communication. The student formats digital information for appropriate and effective communication. | B. Create interactive documents using modeling, simulation, and hypertext. | 9-3, 11-5 |
| 11. Communication. The student delivers the product electronically in a variety of media, with appropriate supervision. | A. Publish information in a variety of ways including, but not limited to, printed copy and monitor displays. | 14-4, U-1 |
| 12. Communication. The student uses technology applications to facilitate evaluation of communication, both process and product. | B. Seek and respond to advice from peers and professionals in evaluating the product. | 36-5, U-1 |
| 12. Communication. The student uses technology applications to facilitate evaluation of communication, both process and product. | C. Debug and solve problems using reference materials and effective strategies. | 14-4, A-1 – U-1 |
| | | |

# Index