Unit 2 – Classes, Methods and Data Fields

An Overview of Digital Circuits and the Binary Number System

Computer Architecture -- An Overview of Digital Circuits

Although computer circuits are extremely sophisticated, they are based on an extremely elementary concept. Computers accomplish almost everything by rapidly switching circuits ***on*** and ***off*** (such circuits are called ***digital circuits***). This seemingly random activity allows computer users to produce spreadsheets, word processing documents, images, animation, Web pages and almost anything else that the human imagination can conceive. It may be difficult at first to understand how switching circuits on and off can accomplish anything at all. Once one appreciates the extreme speed with which the switching is performed, and the idea that ***information is encoded as sequences of these simple "flip-flopping" electrical impulses***, the power of digital circuits becomes clear.

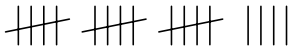
The CPU is the "Engine" of the Computer System

You probably have heard people say that the CPU (central processing unit) is the ***brain*** of a computer system. Despite the widespread use of this idea, however, the CPU is very much ***unlike*** the human brain. It is completely ***devoid*** of any of the higher order abilities that the human brain possesses such as independent thought and reasoning, a sense of consciousness, emotions and the ability to learn from experiences. Considering these rather astonishing capabilities, the electrochemical impulses in the human brain and nervous system are conducted rather slowly (at speeds of up to about 400 km/h).

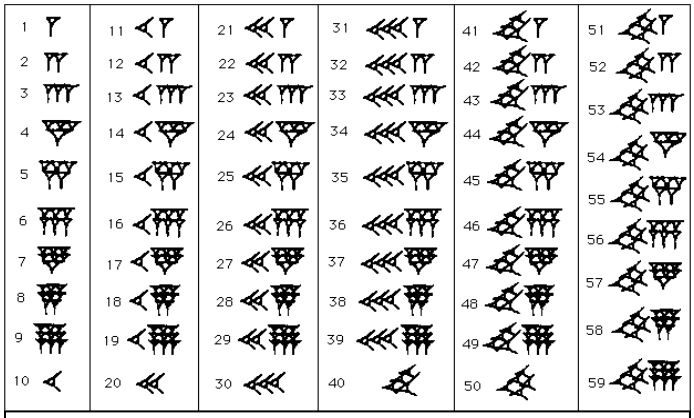
The CPU, on the other hand, is in a sense a "photographic negative" of the human brain. It outperforms the human brain in some ways, but it falls miserably short in many other respects. It can perform arithmetic, arrange numbers from least to greatest and blindly follow instructions at a blazing speed. However, it does not possess any ***cognitive*** powers. The CPU cannot think, understand, interpret or feel. It is merely a rather unintelligent order taker. The latest and most powerful ***processors*** for the home computer market (CPUs are often called processors) can switch circuits on and off at rates of billions of cycles per second. For example, a processor operating with a ***clock speed*** of 3.5 GHz (gigahertz or billions of cycles per second) can switch circuits on and off ***three billion five hundred million*** times per second. In other words, every time the CPU clock ticks, a computer circuit can switch from on to off or vice versa. Considering how dissimilar the CPU and the human brain are, it is far more appropriate to think of the ***CPU as the engine of a computer system.***

Why do Computers Process Binary Numbers and not Decimal Numbers?

Throughout the ages, humans have used various number systems. When primitive humans first started to count, they probably employed the ***unary*** number system, which is ***based on the number one***. If you are familiar with counting with sticks, then you will understand unary numbers. An example is shown below.


This represents the number 19.

Although the unary number system is extremely simple, it has some serious limitations. Imagine representing the number 5983456782 in unary. If you could draw "sticks" at a rate of five per second, it would take almost ***thirty-eight*** years to complete the unary representation of this number. In addition, in the unary number system there is no way of representing zero. Of course, one could attempt to use no sticks to represent zero, however, it would be very difficult to distinguish zero from a blank space.


The 59 symbols used by the Babylonians. These symbols are built from the two basic symbols and , representing one and ten respectively.

Eventually, humans learned how to address these limitations by using number systems that are based on numbers greater than one. An extreme example of this is the ***sexagesimal system*** used by the Babylonian civilization. It was, for religious, mystical or astronomical reasons, based on the number ***sixty***. This means that the Babylonians used fifty-nine different numerals to represent the numbers from one through fifty-nine (there was no symbol for zero). The numbers greater than fifty-nine were represented by using combinations of the first sixty symbols, just as we use combinations of the Hindu-Arabic numerals from zero through nine to represent numbers greater than nine.

If you have ever wondered why there are three hundred and sixty degrees in one revolution, look no further than the Babylonian civilization. The Babylonians believed that one year consisted of three hundred and sixty days. Since one year was viewed as a complete circle, the circle came to be subdivided into three hundred and sixty equal divisions, which we now call degrees. Note that three hundred and sixty is a multiple of sixty, which may help to explain the special status of the number sixty.

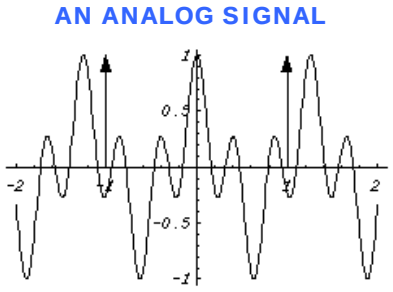
Other civilizations used a variety of different number systems. At one point, the French used an ***octal*** (base eight) number system. The Romans used a ***decimal*** (base 10) number system that was based on Roman numerals, while the Arabs used a decimal system based on Hindu-Arabic numerals. Eventually, most likely for ***anatomical reasons*** (read between the lines here), all humans decided to adopt a decimal number system. Computers, however, use the ***binary*** number system, ***which is based on the number two***.

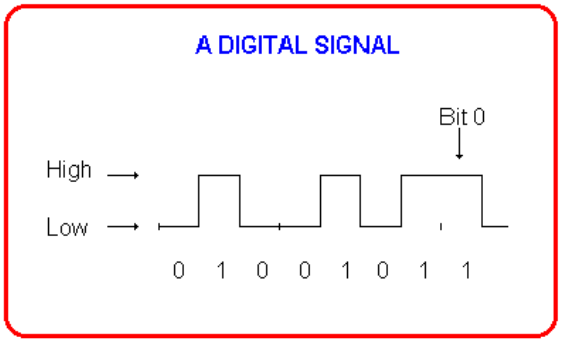
Why Computers Count by "Twos" instead of "Tens" – The Basis of Digital Circuits

If humans count by ten, why then do computers count by two? If you have read the sections on basic computer architecture, the answer to this question should be obvious. If computers accomplish everything by switching circuits on and off, then only two numerals are needed. If ten digits were used instead, a number of electronic complications would arise. The computer circuits would need to be able to detect ten different possible states as opposed to only two. This would increase the probability of errors, the complexity of the circuits and the cost. It is relatively easy, however, to design circuits that detect the difference between "on" and "off." The table given below lists several interpretations of the "on" and "off" states.

Circuit State	*Electronic Representation*	*Binary Representation*	*Logical Representation*
Off	Low Voltage (e.g. 0V)	0	False
On	Higher Voltage (e.g. 5V)	1	True

The true power of a computer system lies in the electronic signals by which communication takes place, that is, the ***digital signals***. Instead of forming a continuous wave pattern like ***analog signals***, digital signals are ***discrete***. This means that they can exist only ***in a finite number of states***, unlike analog signals, which can have an infinite number of states. This allows data to be sent and received in the "language" of ones and zeros, or binary language.





The main advantage of digital signals is that they are so simple! As long as the hardware is functioning correctly and there are no sources of interference, it is always possible to make a ***perfect copy*** of a digital signal because there are only ***two states to detect***, "on" and "off." Analog signals, on the other hand, are impossible to copy perfectly because they are so complex. A copy of an analog signal ***always*** contains some amount of ***distortion*** of the original signal.

Imagine trying to copy a friend's answers on a "true-false" test. It would be very easy to copy your friend's answers perfectly because you only need to distinguish between "T" and "F." Now imagine trying to do the same on an essay test. Even a meticulously careful person with eagle eye vision could not produce an exact copy of his/her friend's essay!

Question

Why is the music industry so concerned about digital copies of audio CDs and the downloading of audio files?

Character Encoding – How Binary Numbers are used to represent Textual Information

A ***character encoding*** (also called a ***character set*** or ***code page***) consists of a code that pairs a sequence of characters from a given set with a sequence of values that can be easily represented on an electronic device (**e.g.** integers, sequences of binary digits, sequences of electrical pulses). This allows text to be stored on computers and transmitted across telecommunication networks. Common examples of character encodings include

- Morse code, which encodes letters of the Latin alphabet as series of long and short depressions of a telegraph key
- ASCII, which encodes letters, numerals and other symbols as sequences of bits (***b***inary dig***its***)

An Old Binary Character Encoding Scheme – Morse Code

Before telephone technology was developed and became widely available, long distance electronic communication was accomplished by using a device called a ***telegraph*** (shown below). Instead of using voice, the telegraph was used to create a series of pulses called "***dots***" and "***dashes***." A dot is created by a quick tap of the telegraph keying device, resulting in a pulse of very short duration. Holding the key down for a longer time, on the other hand, would create a pulse of longer duration known as a dash.

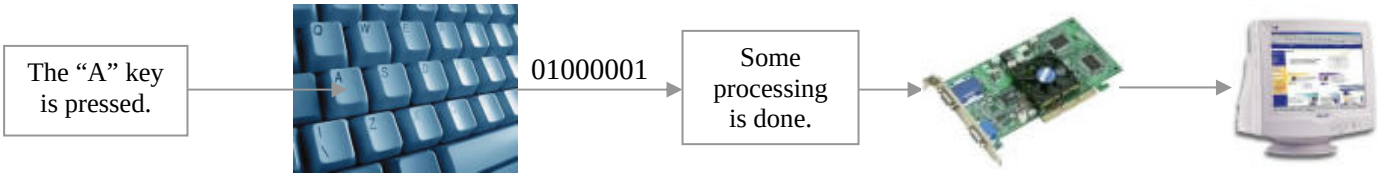
By combining dots and dashes according to the encoding scheme known as ***Morse code*** (shown at the right), messages could be transmitted over long distances. Morse code was also used for long distance radio communication and until recently, was a requirement for obtaining an amateur radio (also known as a "Ham" radio) communications licence.

Morse code key

Letters		Numbers	
A	•—	1	•— — — —
B	—•••	2	•• — — —
C	—•—•	3	••• — —
D	—••	4	•••• —
E	•	5	•••••
F	••—•	6	— ••••
G	— —•	7	— — •••
H	••••	8	— — — ••
I	••	9	— — — — •
J	•— — —	0	— — — — —
K	—•—		
L	•— ••		
M	— —		
N	— •		
O	— — —		
P	•— — •		
Q	— — •—		
R	•— •		
S	•••		
T	—		
U	•• —		
V	••• —		
W	•— —		
X	— •• —		
Y	— •— —		
Z	— — ••		

EBCDIC, ASCII, ANSI, ISO-Latin 1 and other Character Sets

When you press a key on a keyboard, a certain ***binary code*** is transmitted from the keyboard. After some processing, the ***video card*** transmits a signal that causes the character corresponding to the given code to be displayed on the monitor's screen. Of course, this process occurs in such a short time that it appears that the computer is doing something intelligent. All that really happens, however, is that a sequence of "on-off" pulses is transmitted. Using a ***character set*** called "ANSI," for instance, the sequence for the letter "A" is "off, on, off, off, off, off, off, on" or "01000001" in binary form. The process is described pictorially below:



Naturally, each character must have a unique binary code. A collection of such codes is called a ***character set***.

Many different character sets are in use today. One of the first to be used was developed decades ago by IBM. It is called EBCDIC ("Extended Binary- Coded Decimal Interchange Code") and is still in use today in large IBM mainframe computers. In the years since EBCDIC first came on the scene, many other character sets have been developed. While each character set uses a different encoding scheme, they all have one very important element in common. Every character set uses groups of ***bi***nary digi***ts*** or ***bits*** to represent each character. Since many character sets use sequences of eight bits to represent each character, it is convenient to think of such a group as a single unit. A group of eight bits is known as a ***byte***. The table below summarizes some of the most commonly used character sets.

Character Set		Number of Bits in Code	Total Number of Characters that can be Represented	Use/Platform
Abbreviation	Full Name			
ASCII	American Standard Code for Information Interchange	7	$128 = 2^7$	Early Personal Computers
EBCIDIC	Extended Binary-Coded Decimal Interchange Code	8	$256 = 2^8$	IBM Mainframe Computers
ANSI	American National Standards Institute	8	$256 = 2^8$	Windows
ISO-Latin 1	International Standards Organization 8859-1	8	$256 = 2^8$	HTTP and HTML (World Wide Web)
DBCS	Double Byte Character System	16	$65536 = 2^{16}$	Asian Versions of Windows
Unicode	Unicode	Variable (often 16)	Up to $4294967296 = 2^{32}$	Platform Independent

The Relationship between Storage Space and Characters

1 byte	8 bits	1 ANSI/ISO-Latin 1 character
1 kilobyte = 1 KB	1024 bytes	1024 ANSI/ISO-Latin 1 characters
1 megabyte = 1 MB	1024 KB	1048576 ANSI/ISO-Latin 1 characters
1 gigabyte = 1 GB	1024 MB	1073741824 ANSI/ISO-Latin 1 characters
1 terabyte =1 TB	1024 GB	1099511627776 ANSI/ISO-Latin 1 characters

An Example of Unicode Character Mappings

0A80 **Gujarati** **0AFF**

The picture at the right, which consists of characters from the east Indian language Gujarati, shows a small portion of the Unicode character set. Each of the characters in the Unicode character set has a unique ***hexadecimal*** (base 16) code. For example, the hex code of the character ઋ is 0AB3. When converted to 16-bit binary form, 0AB3 is written as 0000101010110011. The actual binary values are not included in the Unicode code charts because they are too long. Hexadecimal form, which is closely related to binary, is far more "human-friendly." (See page CMDF-11 for detailed information on numbers expressed in hexadecimal form.)

Since Unicode consists of various character encodings that use up to 4 bytes per character, it allows for the encoding of far more characters than ANSI. It is very easy to calculate the number of possible binary sequences of a given length *n*:

#binary sequences of length $n = 2^n$

Therefore, in Unicode it is possible to encode up to 2^{32} =4294967296 characters while in ANSI, it is only possible to encode 2^8 =256.

(It is easy to understand why 2^n gives the correct result. In a binary sequence of length *n*, there are ***two choices*** for each position in the sequence. Therefore, the total number of different sequences is $2 \times 2 \times 2 \times \cdots \times 2 = 2^n$.)

Examples

In these questions, assume that ***no data compression techniques*** are used.

(a) How many ANSI characters can be stored on a hard drive that has a storage capacity of 60 GB?

(b) Assuming that the average English word is six characters long, how many English words can be stored on a 60 GB hard drive?

(c) Assuming that the average English novel contains 50000 words, how many English novels can be stored on a 60 GB hard drive?

Solutions

(a) number of bytes = 60(1024)(1024)(1024) = 64424509440 Since one ANSI character uses ***one byte*** of storage, 6442450944 characters can be stored.	(b) number of words = 64424509440 ÷ 6 ≐ 10737418240 Approximately 10737418240 English words can be stored on a 60 GB hard drive.	(c) number of novels = 10737418240 ÷ 50000 ≐ 214750 Approximately 214750 such novels can be stored on a 60 GB hard drive.

Questions

1. Explain the basic principles upon which computer circuits are based.
2. What is a digital circuit?
3. What is a CPU? Explain why CPUs are not intelligent. Why do some people believe that CPUs are intelligent?
4. What is a CPU clock? What is meant by clock speed?
5. If you could write "sticks" at the rate of ten per second, how long would it take to write the unary representation of the number 6734521343?
6. What are the limitations of the unary number system?
7. Why did all humans eventually adopt a decimal number system? What other number systems have been used by other civilizations?
8. Why are there 360 degrees in one full revolution?
9. What are Hindu-Arabic numerals?
10. Why are decimal numbers unsuitable for computer circuitry? Why is the binary system a much better choice?
11. What is a bit? What is a byte?
12. What is a character set?
13. Discuss the various ways in which zero and one are represented at the level of computer circuitry.
14. How many ANSI characters can be stored on a 120 GB hard drive?
15. Convert
    - (a) 209477464 bytes to KB
    - (b) 1.44 MB to KB
    - (c) 209477464 bytes to MB
    - (d) 1.44 MB to bytes
    - (e) 209477464 bytes to GB
    - (f) 147.2 MB to TB
    - (g) 209477464 bytes to TB
    - (h) 1 GB to bits
16. A DVD-ROM disk can store 4.38 MB of data.
    - (a) How many 50000-word novels can be stored on a single CD-ROM disk?
    - (b) Although CD-ROM disks have such an immense storage capacity, why might it be a poor idea to use a CD-ROM disk for the long-term storage of data?
17. In the metric (SI) system of units, the prefix "kilo" means one thousand. Why then, in the world of computers, does "kilo" mean 1024?

Binary, Octal and Hexadecimal Arithmetic

Place Values

As we have learned, computers use the ***binary number system*** for encoding information. Although at first glance binary numbers seem strange and confusing, they operate just like numbers in decimal form. The only difference is that the value of each "place" or "column" is a ***power of two*** instead of a power of ten. The examples given below should help you understand what this means.

Binary Example

Place Values

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

$$= 1\times 2^7 + 0\times 2^6 + 0\times 2^5 + 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0$$
$$= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$$
$$= 155$$

Decimal Example

Place Values

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

$$= 2\times 10^7 + 3\times 10^6 + 1\times 10^5 + 9\times 10^4 + 7\times 10^3 + 8\times 10^2 + 0\times 10^1 + 5\times 10^0$$
$$= 20000000 + 3000000 + 100000 + 90000 + 7000 + 800 + 0 + 5$$
$$= 23197805$$

Octal Example

Place Values

8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0
2	3	1	5	7	0	0	5

$$= 2\times 8^7 + 3\times 8^6 + 1\times 8^5 + 5\times 8^4 + 7\times 8^3 + 0\times 8^2 + 0\times 8^1 + 5\times 8^0$$
$$= 4194304 + 786432 + 32768 + 20480 + 3584 + 0 + 0 + 5$$
$$= 5037573$$

Hexadecimal Example

Place Values

16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
1	9	A	2	E	D	C	F

$$= 1\times 16^7 + 9\times 16^6 + 10\times 16^5 + 2\times 16^4 + 14\times 16^3 + 13\times 16^2 + 12\times 16^1 + 15\times 16^0$$
$$= 268435456 + 150994944 + 10485760 + 131072 + 57344 + 3328 + 192 + 15$$
$$= 430108111$$

As you can see from the above examples, the only significant difference in each case is the ***base*** of the power. The table below summarizes the most important number representation systems for use with computers.

Name	Base	"Digits"
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Various Interpretations of Binary Codes

One of the reasons that programmers need to declare variables and specify their ***types*** is that a particular binary code can be interpreted in many different ways.

The table given below shows how the 16-bit (2 byte) binary number 0000001110100000 can represent two different values.

	Value	
Raw Binary Data	16-bit Signed Integer (**short** Data Type in Java)	Unicode Character (**char** Data Type in Java)
0000001110100000	928	Π

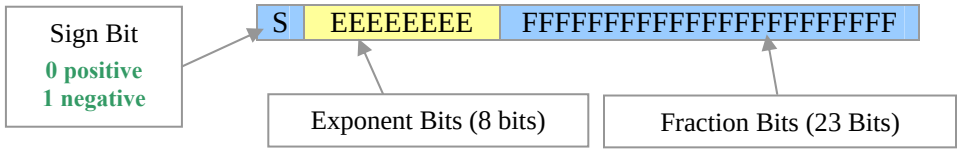
The next table shows how the 32-bit (4 byte) binary number 11000011100110001101000000000000 can represent two different values.

	Value	
Raw Binary Data	32-bit Signed Integer (**int** Data Type in Java)	IEEE 754 Floating-Point Number (**float** Data Type in Java)
11000011100110001101000000000000	–1013395456	–305.625

The IEEE754 Standard for Representing Floating Point Numbers

The IEEE 754 standard allows for 32-bit floating point numbers to be expressed ***correct to seven significant (decimal) digits*** and for 64-bit floating point values to be expressed ***correct to 15 significant (decimal) digits***.

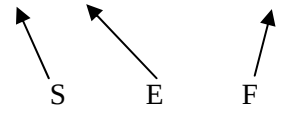
The following format is used to store IEEE 754 32-bit floating point numbers:



The value of the floating point number is taken to be

$$\pm 1.F \times 2^{E-127}$$

The value "1.F" is called the ***mantissa*** or the ***significand***. Note that the ***whole part*** of the mantissa (the most significant bit) is not stored because it always equals 1. The ***fractional part*** of the mantissa, that is "F," is called the ***fraction***. Note that the fraction is expressed in binary form. Also, the ***base*** of the power, which is always 2 due to the use of binary values, is also called the ***radix***. In addition, to avoid storing the sign of the exponent, the exponent is stored in ***biased*** form. That is, the exponent stored is greater than the actual exponent by 127.



$11000011100110001101000000000000 = -1.00110001101000000000000 \times 2^{10000111\,-\,0111111}$ (according to IEEE 754 format)

$= -(1+0\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4}+0\times2^{-5}+0\times2^{-6}+0\times2^{-7}+1\times2^{-8}+1\times2^{-9}+0\times2^{-10}+1\times2^{-11})\times 2^{135\,-\,127}$

$= -(1+.125+.0625+.00390625+.001953125+.000488281) \times 2^{8}$

$= -305.6249999$

$= -305.6250$ (correct to seven significant digits)

More Information

For more information on the Unicode character set, visit http://en.wikipedia.org/wiki/Unicode and http://www.unicode.org. For more information on the IEEE 754 floating point number standard, visit http://en.wikipedia.org/wiki/IEEE_754, http://standards.ieee.org/, and http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html.

The Twos Complement Method of Representing Signed Integers

Example 1 – Positive 8-bit Signed Integers (byte *Data Type in Java*)

Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	This bit pattern represents +91. (The "0" in the sign bit column indicates that the integer is positive.)
0	1	0	1	1	0	1	1	

Example 2 – Negative 8-bit Signed Integers (byte *Data Type in Java*)

The representation of negative integers is a little more complicated than merely changing the sign bit to a "1." To simplify the logic required for binary addition and subtraction, a system called the **"twos complement"** is used. The example given below shows how to convert the negative integer –69 into "twos complement binary form."

1. Convert +69 to 8-bit binary form → **01000101**
2. Invert each bit of the binary number obtained in step 1 → **10111010**
3. Add 00000001 → **10111011**

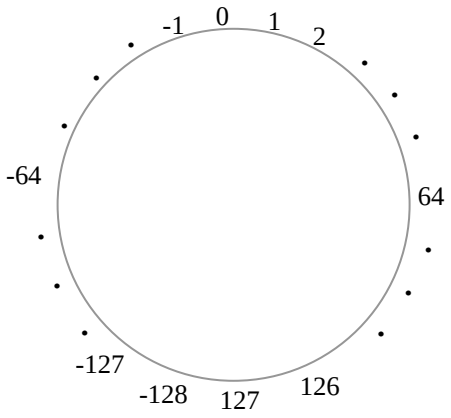
Sign Bit								This bit pattern represents –69. (The "1" in the sign bit column indicates that the integer is negative.)
1	0	1	1	1	0	1	1	

Why use the Twos Complement Method to Represent Signed Integers?

From a human perspective, the twos complement system is rather awkward. When first introduced to this system, a natural question arises in the minds of most students. Why can't we simply change the sign bit from 0 to 1 and be done with it? Why is it necessary to invert the bits and add 1? There are two very good answers to these questions.

1. Simply changing the sign bit from 0 to 1 would give ***two*** different representations of zero, 00000000 and 10000000 because +0 = –0 = 0. This would be both ***confusing*** and ***wasteful***. Instead of having two codes to represent zero, the twos complement system uses the code 10000000 is used to represent –128.
2. Using the twos complement method allows all integer subtractions to be converted to additions. This allows engineers to simplify CPU design because only adder circuits are required to add ***and*** subtract integers.

8-bit Signed Binary Integer	Decimal Value
10000000	–128
10000001	–127
. . .	. . .
11111111	–1
00000000	0
00000001	1
. . .	. . .
01111111	127



The twos complement system (8-bit signed) can be visualized as a circular number line. When the highest value of 127 is reached, adding 1 causes a "wrap-around" back to the lowest value of -128. Thus, adding causes a clockwise movement around the circle.

Exercises

1. Using the specified number of bits, write the binary representation of each of the following integers.
   - **(a)** –32700 (16-bit signed integer, **short** in Java, "Word" in Microsoft lingo)
   - **(b)** –1470987 (32-bit signed integer, **int** in Java, "Dword" in Microsoft lingo)
   - **(c)** 65535 (16-bit unsigned integer, **char** in Java)

2. Interpret the bit pattern 11111111100000000000011000000001 as
   - **(a)** two 16-bit Unicode characters (i.e. two Java **char** values or two 16-bit unsigned integers)
   - **(b)** a 32-bit signed integer (i.e. an **int** value in Java)
   - **(c)** a 32-bit unsigned integer (no equivalent type in Java)
   - **(d)** an IEEE754 32-bit floating point value (a **float** value in Java)

The Importance of Hexadecimal Numbers

Although at the level of a computer's circuitry all information is represented using binary numbers, computer professionals like engineers and software developers very rarely read information in this form. Binary codes are generally far too long to be read and interpreted by humans. The hexadecimal number system, on the other hand, is far easier for humans to understand because large numbers can be represented using a relatively small number of characters. In addition, since 16 is a power of 2 ($2^4 = 16$), it is very easy for a computer to convert from binary to hexadecimal and vice versa.

The table shown below helps you to gain some insight into the usefulness of hexadecimal numbers. It compares the binary and hexadecimal representations of a few integers ranging from 0 to 255.

Decimal	0	10	20	30	40	50	100	200	255
Binary	0	1010	10100	11110	101000	110010	1100100	11001000	11111111
Hexadecimal	0	A	14	1E	28	32	64	C8	FF

As you can see, the hexadecimal representation is the shortest.

Because of this, many computer quantities, such as memory location addresses and colour codes, are often specified using hexadecimal notation. For example, if a computer has 512 MB of RAM installed, the memory locations are numbered as shown in the following table.

Memory Location Address		1 KB	1 MB			512 MB
Decimal Form	0	1024	$1024^2 = 148576$	283547695	...	$512(1024^2) = 512(1048576) = 536870912$
Hexadecimal Form	0	400	100000	10E6982F		20000000

Conclusion

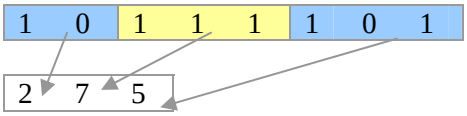
Although digital circuits are based on the binary number system, binary numbers are generally too long to be understood and manipulated easily by humans. The hexadecimal number system, on the other hand, is far more "human-friendly" because large numbers can be represented using a relatively small number of characters. Moreover, since 16 is a power of 2 ($2^4 = 16$), it is very easy to convert from binary to hexadecimal and vice versa. This makes the hexadecimal number system ideal for discussing computer issues such as memory addresses, error codes, colour codes and character set codes. The decimal system (base 10) is not at all suitable because ten is not a simple power of two. As a result, conversions between base 2 and base 10 are more complicated and require greater amounts of processing time.

Converting from one Base to Another

Binary to Octal

1. Starting at the rightmost end of the binary number, form groups of 3 bits.
2. Convert each group of 3 bits to octal form.

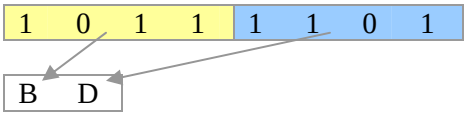
Example 1



Binary to Hexadecimal

1. Starting at the rightmost end of the binary number, form groups of 4 bits.
2. Convert each group of 4 bits to hexadecimal form.

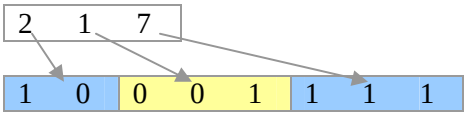
Example 2



Octal to Hexadecimal

1. Convert from octal to binary.
2. Convert from binary to hexadecimal.

Example 3



Now convert 10001111 to hex using the method of example 2.

Octal or Hexadecimal to Binary

Use the method shown in example 3 to convert from octal to binary. To convert from hex to binary, arrange the bits in groups of four instead of three. ***Do not forget to include any leading zeros whenever they are needed!***

Decimal to Binary

As you will soon see, this conversion requires a great deal more processing than the ones shown on the previous page.

Example

Convert $123_{(10)}$ to binary form. Assume that the given value is stored as a Java **byte** value (8-bit signed integer).

Method 1 (Subtraction)

Begin at the left end of the binary number and proceed to the rightmost bit.

S	64	32	16	8	4	2	1
0	1	1	1	1	0	1	1
	123-64 =59	59–32 =27	27-16 =11	11-8 =3	3-0 =3	3-2 =1	1-1 =0

Method 2 (Division by 2)

S	64	32	16	8	4	2	1
0	1	1	1	1	0	1	1

123 ÷ 2 = 61 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)
61 ÷ 2 = 30 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)
30 ÷ 2 = 15 R 0 ∴ binary number ***ends*** in 0 (i.e. it's even)
15 ÷ 2 = 7 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)
7 ÷ 2 = 3 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)
3 ÷ 2 = 1 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)
1 ÷ 2 = 0 R 1 ∴ binary number ***ends*** in 1 (i.e. it's odd)

Elementary School Arithmetic Revisited

Although most of us have known algorithms for adding and subtracting since elementary school, few of us understand why these algorithms produce correct answers. The following examples should help you understand why the steps that you were taught in elementary school actually work!

Decimal Examples

1000	100	10	1
1	1	1	
	9	8	7
+	8	9	6
1	8	8	3

7 ones + 6 ones = 13 ones
= 1 ten + 3 ones
1 ten + 8 tens + 9 tens = 18 tens
= 1 hundred + 8 tens
1 hund. + 9 hund. + 8 hund. = 18 hund.
= 1 thousand + 8 hund.

1000	100	10	1
0	9	9	10
~~1~~	~~0~~	~~0~~	~~0~~
–	6	4	3
	3	5	7

1 thousand is ***borrowed*** from the thousands column
1 thousand = 9 hundreds + 10 tens + 10 ones

Binary Examples

8	4	2	1
1	1	1	
	1	1	1
+	1	0	1
1	1	0	0

1 one + 1 one = $2_{(10)}$ ones
= 1 two + 0 ones
1 two + 1 two + 1 two = 3 twos
= 1 four + 1 two
1 four +1 four + 1 four = $3_{(10)}$ fours
= 1 eight + 1 four

8	4	2	1
0	1	1	10
~~1~~	~~0~~	~~0~~	~~0~~
–	1	1	1
			1

1 "eight" is ***borrowed*** from the "eights" column
1 eight = 1 four + 1 two + $10_{(2)}$ ones

Recall that this means "$2_{(10)}$" in binary.

Octal Examples

512	64	8	1
1	1	1	
	7	6	4
+	7	4	7
1	7	3	3

4 ones + 7 ones = $13_{(8)}$ ones
= 1 eight +3 ones
3 "8s" + 6 "8s" + 4 "8s" = $13_{(8)}$ eights
= 1 "64" + 3 "8s"
1 "64" +7 "64s" + 7 "64s" = $17_{(8)}$ "64s"
= 1 "512" + 7 "64s"

512	64	8	1
5	7	14	13
~~6~~	~~0~~	~~5~~	~~3~~
–	4	7	7
5	3	5	4

1 "8" is ***borrowed*** from the "8s" column
3 "ones" + 1 "eight" =$13_{(8)}$ ones
1 "512" is ***borrowed*** from the "512s" column
1 "512" + 4 "8s" =7 "64s" + $14_{(8)}$ "8s"

Hexadecimal Examples

4096	256	16	1
1	1	1	
	A	C	E
+	7	F	9
1	2	C	7

E ones + 9 ones = $17_{(16)}$ ones
= 1 "16" +7 ones
1 "16" + C "16s" + F "16s" = $1C_{(16)}$ "16s"
= 1 "256" + C "16s"
1 "256" +A "256s" + 7 "256s" = $12_{(16)}$ "256s"
= 1 "4096" + 2 "256s"

4096	256	16	1
E	F	1D	10
~~F~~	~~0~~	~~E~~	~~0~~
–	4	F	7
E	B	E	9

1 "16" is ***borrowed*** from the "16s" column
0 "ones" + 1 "16" =$10_{(16)}$ ones
1 "4096" is ***borrowed*** from the "4096s" column
1 "4096" =F "256s" + $10_{(16)}$ "16s"

Exercises and Problems

1. Evaluate without using a calculator. Show all carrying or borrowing!

$11001100111_{(2)} + 10111100011_{(2)}$

$11001111111_{(2)} + 10111101111_{(2)}$

$A3FAE_{(16)} + 9FFBC_{(16)}$

$7654321_{(8)} + 6775322_{(8)}$

$11011101111_{(2)} + 11011100111_{(2)} + 11001111111_{(2)}$

$FFFBC_{(16)} + A3FAE_{(16)} + 9FFBC_{(16)}$

$11001100111_{(2)} - 10111100011_{(2)}$

$11001111111_{(2)} - 10111101111_{(2)}$

$A3FAE_{(16)} - 9FFBC_{(16)}$

$7654321_{(8)} - 6775322_{(8)}$

2. Explain why a "1" is borrowed from the "1024" (2^{10}) place. In addition, explain how this "1" is redistributed to the other places.

$$10000000000_{(2)} - 1_{(2)}$$

3. Now that you know how to add and subtract using numbers that are expressed in non-decimal form, try the following multiplication questions.

$10011_{(2)} \times 10111_{(2)}$

$3AE_{(16)} \times 9FC_{(16)}$

$765_{(8)} \times 677_{(8)}$

4. Computers ***do not*** perform binary subtraction in the manner described above (i.e. using borrowing). Instead, the adder circuits are used to "add the negative." This avoids having to design circuitry for subtraction as well as addition. The following steps are used to perform the subtraction $a - b$:

i. Express $-b$ in the "twos complement" binary form.

ii. Perform $a + (-b)$

Perform the following subtraction using the method described above. Use the method of "borrowing" to verify that both methods produce the same answer. Assume that the integers are stored in 8-bit signed form (e.g. **byte** in Java).

Recall that the leftmost bit is the "sign" bit.

$11101001_{(2)} - 00100101_{(2)}$

5. Convert each of the following to decimal form. (Assume that no sign bits are used.)

(a) $110110100111_{(2)}$ (b) $74675_{(8)}$ (c) $FACE_{(16)}$

6. Convert each of the following to binary form

(a) $74675_{(8)}$ (b) $FACE_{(16)}$ (c) $32452_{(10)}$

7. Convert each of the following to octal form

(a) $110110100111_{(2)}$ (b) $FACE_{(16)}$ (c) $34512_{(10)}$

8. Convert each of the following to hexadecimal form

(a) $110110100111_{(2)}$ (b) $74675_{(8)}$ (c) $74675_{(10)}$

Assignment: Exploring Primitive Data Types in Java

KU	APP	COM
/19	/21	/10

You have already encountered various primitive Java data types (e.g. **int**, **short**, **float**, **char**)[1]. In this assignment, you will have a further opportunity to obtain and demonstrate a better grasp of their various representations and interpretations. **Note:** There will be up to 10 communication marks awarded for the clarity of your answers.

1. Complete the following chart. **(9 KU)**

Data Type	Description	Size/Format	Range
byte	Byte-length signed integer	8-bit twos complement	–128 … 127
long			
double			
boolean			

Once you have completed the chart, you should understand both what these types are ***and*** their structure.

2. Choose 2 different negative integers which can be converted into (signed) ***8-bit twos complement form***. Convert them. Show your work. Do it cleanly and concisely! **(6 APP)**

Examples
Negative Decimal Integer: –89
Positive Binary Representation: 01011001 (representation of +89)
Bit Inversion: 10100110
Addition of 1: 10100111

1)

2)

1 See http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html for additional reference.

3. a) Does the **double** data type use the twos complement method to represent negative numbers? **(1 KU)**

b) Explain how 64 bits of raw binary data can be interpreted as a floating point number. Note that for this question you will need to do some research on the 64-bit version of IEEE 754. **(9 KU)**

c) Can a **double** type (64 bits) represent values greater than 2^{64}? Why or why not? **(5 APP)**

d) Choose a floating-point value greater than or equal to 3.5E38 (3.5×10^{38}) and show how it would be represented in IEEE 754 64-bit format. **(10 APP)**

Basic Class Structure

```
//Beginning of class declaration
public|private|protected [abstract] [final] [static] class ClassName
                                        [extends SuperClassName | implements InterfaceName]
{
      //DATA FIELDS should be declared first.  As with all members of a class, the data
      //fields should be public, private or protected.  Public data fields are visible
      //everywhere their corresponding classes are visible. A protected data field is
      //only visible within its class, within its subclasses or within the class package.
      //Note that the subclasses may reside within different packages. Using the private
      //modifier in the declaration of a data field hides it in such a way that it cannot
      //be directly referenced outside of the class in which it is declared.
      public|private|protected [final] [static] type dataFieldName [=initialValue]; //variable
```

Exactly one of these modifiers *must be* used. (The "|" symbol means "or.")

These modifiers *are optional*. If "**final**" is used, then "**static**" should also be used because it is wasteful to create instances of constants! (The "[ ]" mean "optional.")

Data type of variable. (e.g. **int**, **float**, …)

Name of variable.

Initial Value of variable. (Optional)

```
      public|private|protected [final] [static] Class dataFieldName [=new callToConstructor(…)];//object
```

Class Name (e.g. `String`, `Array` …)

Name of object.

Call to *Constructor Method*

```
      //CONSTRUCTOR METHODS, which are used to initialize objects, should be declared
      //next.  Note that constructor methods ARE NOT considered members of a class.
      //Therefore, even if a class is derived from (i.e. is a subclass of) an existing
      //class, it must have its own set of constructor methods. Note that the name of
      //each constructor method MUST MATCH the name of the class. Note also that
      //constructors may be overloaded.
      public ClassName(parameterList)
      {
          //Constructor code goes here.
      }

      //Finally, all other methods should be declared using the format described below.
      //Note that the instance methods of a given class may override instance methods
      //with the same signature in any of its superclasses. In addition, static (class)
      //methods of a given class may hide static methods with the same signature in any
      //of its superclasses. Like constructors, methods may be overloaded.
      public|private|protected [abstract] [native] [synchronized] [final] [static] returnType
                                        methodName(parameterList) [throws Exception1, Exception2, …]
      {
      }
}//End of class declaration
```

Exactly one of these modifiers *must be* used. (The "|" symbol means "or.")

These modifiers *are optional*. The keywords "**final**" and "**static**" have the same meaning as with data fields. The other keywords are usually used only in advanced applications. Note that not all combinations of these keywords are allowed. See chapter 8 of the "Java Language Specification" for more information.

A "**throws**" clause is also *optional*. It is used to list any exceptions that a method can throw that are ***not*** derived from the classes "Error" or "RuntimeException." The method can contain a "**try** … **catch**" block for detecting and possibly handling any caught exceptions.

Understanding Classes and Objects at an Intuitive Level

Why use Classes?

- A class can be used as a template for modelling and creating objects. (**e.g.** "Form" class)
  - ⇒ We DO create instances of such classes by using the "**new**" keyword and calling a constructor.
- Classes can also be used as a convenient structure for storing related methods and data fields (**e.g.** "Math" class)
  - ⇒ We DO NOT create instances of such classes.

The "Automobile" Class

We shall design a class that models a real-world automobile. For the sake of simplicity and brevity, we shall focus only on certain key characteristics of car objects.

Data Fields

private String **colour**
Stores the colour of the automobile as a string.

private float direction
Stores the direction in which the car is moving (in degrees, from 0 to 360).

private String **engine**
Stores the type of engine as a string.

private float engineSize
Stores the engine size (in Litres) as a **float** value.

private float fuelCapacity
Stores the fuel capacity (in Litres) as a **float** value.

private byte gear
Stores the gear that is engaged in the transmission (1, 2, 3, 4, etc. for forward gears, -1 for reverse and 0 for neutral).

private String **make**
Stores the make of the automobile as a string.

private String **model**
Stores the model name of the automobile as a string.

private float power
Stores the maximum power (in kW).

private float torque
Stores the maximum torque (in Nm).

private float speed
Stores the speed of the car (in km/h).

Write at least three more data fields for the `Automobile` ***class.***

Constructor Methods

Automobile()
Sets the initial values of the data fields to default values. Numeric fields are set to 0 and String fields are set to "".

Write at least three more constructors for the `Automobile` ***class.***

Other (Public) Methods

void accelerate(float, float)
Gradually changes the speed of the car from the current value to value specified by the first **float** argument. The second **float** argument specifies the rate of acceleration that should be used.

String **getColour()**
Returns the value of the "colour" data field.

float getDirection()
Returns the value of the "direction" data field.

String **getEngine()**
Returns the value of the "engine" data field.

float getEngineSize()
Returns the value of the "engineSize" data field.

float getFuelCapacity()
Returns the value of the "fuelCapacity" data field.

byte getGear()
Returns the value of the "gear" data field.

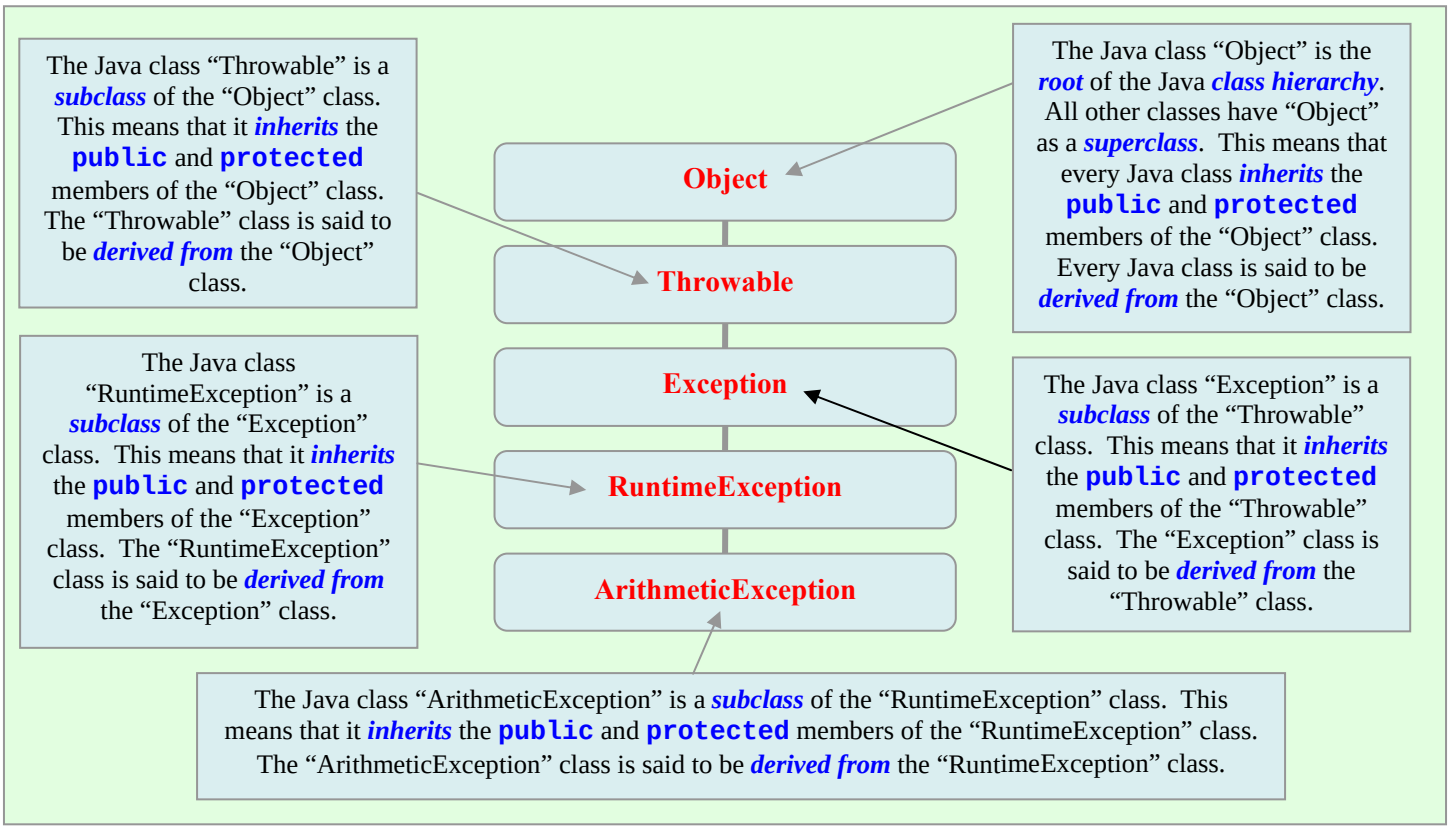
String **getMake()**
Returns the value of the "make" data field.

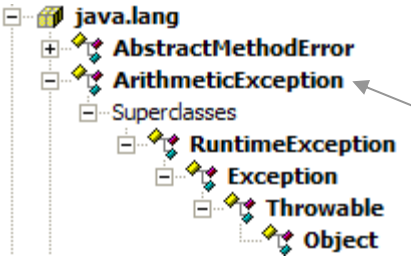
String **getModel()**
Sets the value of the "model" data field.

Write at least three more public methods for the `Automobile` ***class.***

Class Hierarchies and Inheritance

Example of Class Inheritance





The class "ArithmeticException" is found within the "java.lang" package. The J++ object browser can be used to show the hierarchical relationships of the "ArithmeticException" class.

Note
Constructor methods are ***not*** considered members of a class. Therefore, they are not inherited by subclasses. Each subclass must have its own constructor methods.

The "`extends`" Keyword

The "**`extends`**" keyword is used whenever you would like to create a class that is based on an existing class. Whenever you create a form in J++, you are actually defining a new class based on the "Form" class.

e.g. `public class FormTest extends Form`

By using the "**`extends`**" keyword, the "FormTest" class inherits all the **`public`** and **`protected`** members of the "Form" class. The advantage of this is that it is easy to build new classes that are built upon the foundation of existing classes, thereby eliminating the problem of "reinventing the wheel."

Note that it is ***not*** possible to extend all classes. Some classes, such as the String class, are defined as "**`final`**," which means that they cannot be used as the basis for a new class.

Question

What is the difference between a **`public`** and a **`protected`** member of a class?

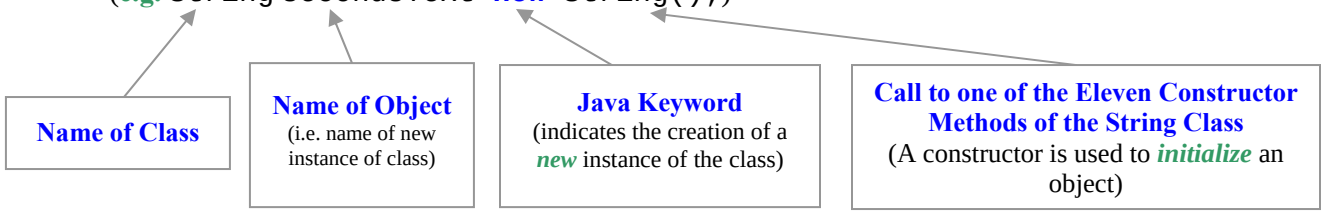
Using Time Converter 1.1 to Review Unit 1

Concepts Introduced and/or Reviewed in Time Converter 1.1

Studying the "Time Converter 1.1" program is a great way to review many of the concepts introduced in unit 1. You will find all the necessary files for "Time Converter 1.1" in the folder **I:\Out\Nolfi\Ics4mo\Time Converter 1.1** (or you can download all programming examples). In order for this program to work correctly, you must copy the "Time Converter 1.1" folder to your **g:** drives. Do not try to run the program directly from the "I" drive!

The "Time Converter 1.1" program is designed to ***introduce*** or ***review*** the following concepts:

1. ***Declaring a variable*** (e.g. `long secondsElapsed;`)
2. ***Declaring and initializing a variable*** in the same statement (e.g. `long secondsElapsed=0;`)
3. ***Declaring objects*** (e.g. `String secondsText;`). Note that in this statement, "String" is the name of the ***class*** and "secondsText" is the name of the ***object***. In this example, an ***instance*** of the class (i.e. the object) has not yet been created. All that has occurred is that the name "secondsText" has been associated with a yet-to-be-created object.
4. ***Declaring an object and instantiating*** (creating an instance of) ***a class*** in the same statement
   (**e.g.** `String secondsText=new String();`)



5. ***Importing*** packages and classes.
6. The "**`extends`**" keyword (used to create a new ***subclass*** that inherits the *fields* and *methods* of its ***superclass***)
7. ***Cast*** operator (This is used to ***coerce*** (force) type conversions. **e.g.** `char keyPress=(char)0;`)
8. Using the ***ternary conditional operator*** "?:"
9. Exception handling using a **`try…catch…finally`** structure
10. Java ***operators*** and ***primitive data types***
11. ***Event handling*** (including the routing of events for a group of objects to same event handler)
12. ***Classes***
13. Using the Java ***String*** class
14. Defining ***methods***
15. Creating ***menus*** in J++
16. Creating ***keyboard shortcuts*** for menus in J++
17. Using **"Timer"** controls in J++
18. Creating J++ programs that use more than one form (***owner forms*** and ***owned forms***)
19. Using **"ColorDialog"** controls in J++
20. Using **"Rich Edit"** controls in J++
21. Using the Java **"Date"** class (`java.util.Date`)
22. Using ***arrays*** whose elements are ***objects*** (e.g. `Color[] colorAttribute = new Color[3];`)
23. ***Class fields*** (also called **`static`** ***fields***) versus ***instance fields*** (non-static fields)
24. ***Class methods*** (also called **`static`** ***methods***) versus ***instance methods*** (non-static methods)
25. ***Modifiers*** (e.g. **`public, protected, private, static, final`**)
26. The **`return`** keyword
27. ***Encapsulation***
28. ***Information Hiding***
29. ***Inheritance***
30. ***Overriding (Single Polymorphism)***
31. ***Overloading and Hiding***

Putting all this Knowledge into Practice – Binary Blaster

Description of the "Binary Blaster" Project

Working as a team, our class shall develop software that can perform the following functions:

1. Convert integers expressed in binary/octal/decimal/hexadecimal form to binary/octal/decimal/hexadecimal form. Negative integers should be expressed in twos complement form.

2. Add and subtract integers expressed in binary/octal/decimal/hexadecimal form.

3. Given a colour code expressed in hexadecimal form, display the colour (24-bit colour).

4. Given a Unicode character code expressed in hexadecimal form, display the character.

5. Given a Unicode character entered using the keyboard or pasted from another program, display its Unicode code in both hexadecimal and decimal forms.

6. Given 32 or 64 bits of raw binary data, display the equivalent IEEE754 value. (See http://en.wikipedia.org/wiki/IEEE_754, page 9 and **I:\4Students\OUT\Nolfi\Ics4m0\ieee754** for more information.)

7. Use the ***simple method of data encryption described below*** to send and receive encrypted messages. Your program should be able to encode an unencrypted text message as well as decode an encrypted text message.

8. Use the ***simple method of data recovery*** (i.e. error correction) ***described below*** to check messages for ***random spontaneous bit inversion errors*** (i.e. a "0" is changed spontaneously to a "1" or a "1" is changed spontaneously to a "0."). Random spontaneous bit inversion errors are known to be caused by cosmic radiation, lightning and many other sources of interference ("noise").

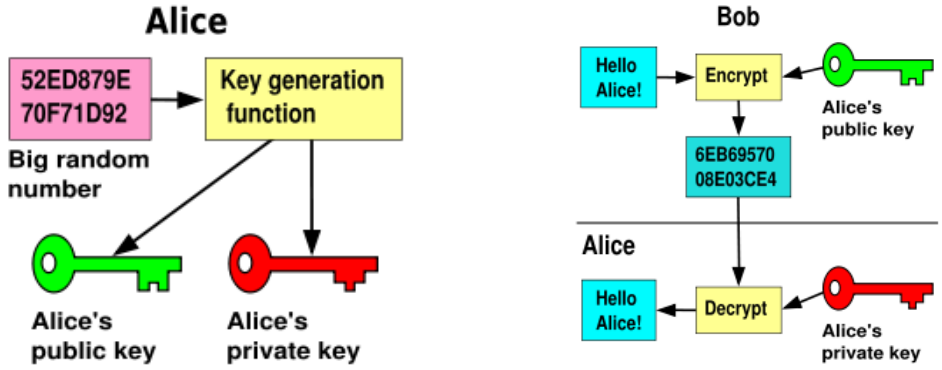
9. Use the simple method of ***data compression described below*** (called ***run-length encoding***) to reduce the amount of storage space required for files containing plain text (text stored in Unicode format).

Each student will develop a specific portion of this software. A project manager will collect the individual parts and assemble them into a working program.

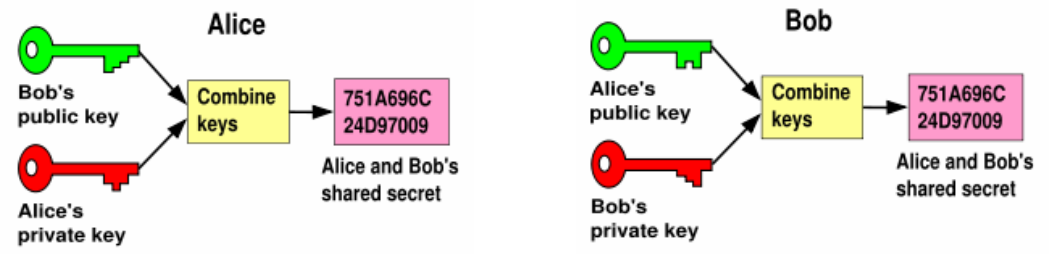
Note

1. The method of ***data encryption*** described on page 24 is much simpler and much less effective than the methods used in practice. For example, for secure Web connections (i.e. "https"), 128-bit (or greater) encryption is used. Examples of such encryption algorithms include ***TwoFish, BlowFish, Serpent, RC6, MARS*** and ***Rijndael***.

   Such methods are either ***asymmetric*** or ***symmetric***. ***Asymmetric methods*** use a "public-key/private-key" pair, as illustrated in the following diagrams.



   Symmetric methods use a shared private key as illustrated in the following diagrams.



2. The method of ***data recovery*** described on page 25 is also much simpler and much less effective than those used in practice. The most commonly used methods are called ***checksum algorithms***. Examples of these include ***CRC-8***, ***CRC-ARC***, ***CRC-16*** and ***CRC-32*** ("CRC" stands for ***cyclic redundancy check***). ***Cryptographic hash functions*** are related to checksums but include additional security features.

3. Not at all surprisingly, the method of ***data compression*** described on page 26 is very crude and ineffective compared to the methods used in practice. The most commonly used ***lossless*** methods of compression are variants of the ***LZ algorithm***. ***MPEG*** (which includes both ***JPEG*** and ***MP3***) is a ***lossy*** method of compression that is used to compress movie files.

Simple Method of Data Encryption

Using a 5-bit binary code, it is possible to encode all the letters of the alphabet and a few punctuation symbols as shown below.

A 00000	I 01000	Q 10000	Y 11000
B 00001	J 01001	R 10001	Z 11001
C 00010	K 01010	S 10010	! 11010
D 00011	L 01011	T 10011	? 11011
E 00100	M 01100	U 10100	. 11100
F 00101	N 01101	V 10101	, 11101
G 00110	O 01110	W 10110	" 11110
H 00111	P 01111	X 10111	space 11111

Using this system, the ***first three words*** of the message "DO YOUR WORK OR I'LL WRAP THIS KEYBOARD AROUND YOUR NECK!" would be encoded as follows:

D	O		Y	O	U	R		W	O	R	K
00011	01110	11111	11000	01110	10100	10001	11111	10110	01110	10001	01010

Exercise

Using the table of codes given above, complete the binary representation of the message given above.

The Encryption Scheme Requires a Special form of Binary Addition and a "Private Key"

Anyone who has knowledge of this encoding scheme, however, can intercept messages and read them. Even without knowledge of the scheme, it is possible to write computer programs that scan raw binary data and look for patterns that can help to match the data to known words.

To ensure the privacy of the information, therefore, it is necessary to ***encrypt*** the message. Although it is extremely difficult to design an encryption method that is "uncrackable," encrypting data greatly reduces the probability that sensitive information will be read by unauthorized parties.

For the purposes of this assignment, we shall use a simple method of encryption that is easy to implement. (It is, however, a ***weak method of encryption***, meaning that it is relatively easy to crack.) In this method, ***addition*** and ***subtraction*** are defined as follows:

0+0=0 0+1=1 1+0=1 1+1=0	0-0=0 0-1=1 1-0=1 1-1=0	Notice that addition and subtraction are identical! Also note that there is ***no carrying or borrowing***, as there is in true binary addition and subtraction.

In addition to the operations defined above, this encryption method requires a 5-bit code known as a ***private key***. The key is known to the sender and the recipient but not to anyone else. To encrypt a message, the key is used, along with the special operations defined above. In the following example, the private key 10101 is used to encrypt the message "DO YOUR WORK."

Original Message in Text Form	D	O		Y	O	U	R		W	O	R	K
Original Message in Binary Form	00011	01110	11111	11000	01110	10100	10001	11111	10001	01110	10001	01010
"Add" Private Key	+10101	+10101	+10101	+10101	+10101	+10101	+10101	+10101	+10101	+10101	+10101	+10101
Encrypted Message in Binary Form	10110	11011	01010	01101	11011	00001	00100	01010	00100	11011	00100	11111
Encrypted Message in Text Form (Message sent to recipient)	W	?	K	N	?	B	E	K	E	?	E	

To ***decrypt*** the message, simply reverse the above steps.

Encrypted Message in Text Form	W	?	K	N	?	B	E	K	E	?	E	
Encrypted Message in Binary Form	10110	11011	01010	01101	11011	00001	00100	01010	00100	11011	00100	11111
"Subtract" Private Key	–10101	–10101	–10101	–10101	–10101	–10101	–10101	–10101	–10101	–10101	–10101	–10101
Decrypted Message in Binary Form	00011	01110	11111	11000	01110	10100	10001	11111	10001	01110	10001	01010
Decrypted Message in Text Form	D	O		Y	O	U	R		W	O	R	K

Simple Data Recovery Method

When digital signals are in transit between a point of transmission and a point of reception, they are subject to ***spontaneous bit inversion errors***. This means that a "1" can be spontaneously changed to a "0" and a "0" can be spontaneously changed to a "1." If the digital signals are transmitted using radio waves (i.e. wireless data transfers), then they are especially susceptible to such "interference." Sources of interference that are know to cause spontaneous bit inversion errors include cosmic radiation (radiation that comes from space), lightning, power surges and strong electromagnetic fields.

Fortunately, computer scientists have developed algorithms that allow spontaneous bit inversion errors to be detected and even corrected. A very simple such method, used originally by NASA, is able to correct one-bit errors. (It cannot correct errors of two or more bits.) It is based on the same type of addition used in the simple encryption method describe above, as well as the vector operations ***vector addition*** and ***dot product of two vectors***.

Description of a One-Bit Correction Method

This method relies on a vector operation known as the ***dot product***. The example below shows how to apply the dot product to two 7-dimensional binary vectors (using the special form of addition defined above).

$$\begin{aligned}&(1001101)\bullet(0010111)\\ &= 1(0)+0(0)+0(1)+1(0)+1(1)+0(1)+1(1)\\ &= 0+0+0+0+1+0+1\\ &= 0\end{aligned}$$

Number of Bits in Code	Error Correcting Matrix	Fifteen Possible Data Vectors	Error Correcting Property	Interpretation of Received Data Vector
7	$\begin{pmatrix}1001101\\0101011\\0010111\end{pmatrix}$	0001110 1001101 0010111 1010100 0011001 1011010 0100101 1100110 0101011 1101000 0110010 1110001 0111100 1111111 1000011	Each of the fifteen data vectors has the property that ***its dot product with each row of the error correcting matrix must be zero***. When a data vector is received, it is "dotted" with each row of the error correcting matrix.	The received data vector is "dotted" with each row of the error correcting matrix. If each dot product is zero, the received vector is considered correct. If not, the error can be corrected (if only one bit is incorrect) by using the columns of the error correcting matrix (see example below).

Suppose that the vector (1011010) is transmitted from a spacecraft orbiting Mars. As the signal travels toward the Earth, it passes through a stream of highly energetic particles from the solar wind, which causes a spontaneous bit inversion error. The vector received at an Earth monitoring station is (1010010). By "dotting" this received vector with each row of the error correcting matrix, we obtain

$$\begin{aligned}(1001101)\bullet(1010010) &= 1\\ (0101011)\bullet(1010010) &= 1\\ (0010111)\bullet(1010010) &= 0\end{aligned}$$

Since we have at least one dot product that is not zero, we know that an error has taken place. Where is the error? Notice that the column vector $\begin{pmatrix}1\\1\\0\end{pmatrix}$ matches the ***fourth*** column of the error correcting matrix. Therefore, we know that the fourth bit of the received vector must be wrong.

Simple Method of Data Compression – Run-Length Encoding (RLE)

Run-Length Encoding is a very simple method of ***lossless*** data compression. It forms the basis of the ***GIF*** data compression algorithm and is also used in the final stages of more sophisticated algorithms such as ***JPEG***. This method of compression involves searching for runs of consecutive data and storing the number of repetitions of the data. For example, the string "aaaaaaa" could be stored as "7a" since there are 7 repetitions of the character "a." Similarly, the string "abcddddddcbbbbabcdef" could be stored as "abc6dc4babcdef."

Obviously, this very simple scheme is not of much practical value because it does not allow for the encoding of numbers. Since the digits from 2 to 9 are used to represent the length of a run of consecutive characters, it is not possible to distinguish between a number that is used for this purpose and one that is actually part of the data. To overcome this obstacle, we can use a slightly modified version of the scheme described above.

Rules for Simple Version of RLE for this Project

1. Any sequence of two to nine identical characters is encoded by using two characters. The first character is the length of the sequence, represented by one of the characters "2" through "9." The second character is the value of the repeated character. If a sequence consists of more than nine identical characters, each group of nine characters is encoded separately.
2. Any sequence of characters that does not contain consecutive repetitions of any characters is represented by a "1" character followed by the sequence of characters and terminated with another "1." If a "1" appears as part of the sequence, it is preceded with a "1." In this case, the first "1" acts as an ***escape character*** (in the same way that the "\" is used as an escape character in C, C++ and Java).

Using this method, the string "abc11111ddddddddddddddddcbbbbabc234444def,,,,510234" would be stored as follows

1abc1519d7d1c14b1abc231541def14,151102341

These "1's" are used to mark the beginning and end of a sequence that does not contain consecutive repetitions of characters.	Numbers from 2 to 9 (shown in blue) are used to store the length of sequences of identical characters.	In this case, the "1" is used as an ***escape character***, not as a ***delimiter***. It indicates that the "1" that follows it is part of the data.

STOP! Do NOT Write Any Code Yet! This is a big Project and Requires a Great Deal of Planning!!

By this stage in your programming education, it *should be* deeply engrained in your grey matter that you should not write any code for a project of this scope until you first complete ***several extremely important preliminary steps***. The following is a list of ***strongly recommended steps*** that you should follow ***before*** you attempt to write ***even a single line of code!***

1. First you must ensure that you understand fully all the problems that need to be solved. In addition, it is essential that you understand and are able to apply what we have learned about binary numbers. To ensure that you have a good understanding, reread pages 3 – 13 and answer ***all questions***. In addition, answer the supplementary questions given below.

(a) What is lossless data compression? What is lossy data compression? Give examples of each.

(b) List all the escape sequences used in C, C++ and Java. (For example, the new line character is represented as `'\n'` and `'\0'` represents the string terminating character.)

(c) What is the maximum compression ratio achievable with the simple method of RLE described above? What is the minimum compression ratio achievable?

2. Once step one is completed, you should begin ***planning*** the user interface for your program. ***Do not begin creating*** the user interface until you have a g***ood design laid out on paper***. While planning your interface, keep in mind that the program will have a large number of features that are in some way related to binary numbers but not necessarily closely related to one another. ***Give a great deal of thought to how you will integrate all these features into one piece of software in a coherent and unified manner.***
3. Show your design for your user interface to your classmates, friends and of course, yours truly, Mr. Nolfi. These people should provide you with objective feedback about your interface.
4. Once you are satisfied with your interface design, use the Visual J++ form editor (and any other tools that you require) to create your interface. During this process, ensure that all objects are named in a descriptive manner and that all object names adhere to the conventions that we have used throughout the course.
5. Break up the large problems to be solved into a series of smaller and simpler sub-problems. A "block diagram" is very useful for this step.
6. Begin writing pseudo-code for the algorithms that you will be using to solve all the sub-problems. During this process, always remember to give thought to possible ways of improving the algorithms that you have chosen.
7. Design the ***class structure*** of your program. Each student will be given a copy of the `SuperStringMethods` class to assist in the processing of strings. What other classes will you need to design? What will be their structure? What methods and/or data fields will they contain?
8. Design the ***file structure*** of your program. (More details to follow)
9. Once steps one to eight have been completed to the best of your ability, you may begin to write code. You should focus on ***one method at a time***. Before you add a method to a class, test it fully in isolation. Once you are confident that the method is error free, you may add it to the class and move on to the next method.
10. While you write your code, remember to adhere to all the positive programming practices that we have been discussing since grade ten.
    - □ ***Indent all code properly.***
    - □ ***Use descriptive, meaningful names and follow all naming conventions.***
    - □ ***Document (comment) all methods clearly and accurately. Each method should be introduced by a comment that explains the purpose of the method and the meaning/purpose of each parameter.***
    - □ ***Document all abstruse (difficult-to-understand) lines of code.***
    - □ ***Do not document any self-explanatory lines of code.***
    - □ ***Insert blank lines in strategic places to prevent the code from having a sloppy and cluttered appearance.***
    - □ ***Include exception handling to ensure that your program will behave gracefully even when the user doesn't.***
11. Test your software rigorously. Ensure that it performs well under a variety of different conditions. Do not forget to test the ***boundary/extreme cases!*** It is essential that you allow other people, especially non-programmers, to test your software!
12. If you have completed steps 1 – 11 in a highly proficient manner, you deserve a break! Have a coffee, eat a doughnut and watch "The Simpsons." While watching "The Simpsons," continuously repeat the phrase "I never want to become like Homer" until the episode is over! You may also need some well-deserved sleep!

Binary Blaster Master Plan

The heart of Binary Blaster will be a class called “NumericString.” Since our software will input numbers in string form, it will be necessary to design a variety of methods that can manipulate numbers that are stored as strings.

```
/**
 * The following is a template for the "NumericString" class. All required data fields and
 * constructors are given. However, only the signatures of most of the remaining methods are given.
 * You will write the code to implement the static methods and instance methods for which code is
 * not given.
 */
public class NumericString
{
   /**
    * CONSTANT DATA FIELDS
    * In keeping with the principle of ENCAPSULATION, the data fields (both constant and variable)
    * are bundled with the methods that operate on the data. This allows for the construction
    * of logical, cohesive structures.
    */
   public final static byte BYTE_SIZE=8, SHORT_SIZE=16, CHAR_SIZE=16, INT_SIZE=32, LONG_SIZE=64;
   public final static byte FLOAT_SIZE=32, DOUBLE_SIZE=64, COLOR_SIZE=24;
   public final static byte BINARY=2, OCTAL=8, DECIMAL=10, HEX=16;
   public final static short IEEE754=754;
   /**
    * VARIABLE DATA FIELDS
    * In keeping with the principle of INFORMATION HIDING, the variable data fields are declared
    * as "private." This prevents them from being exposed needlessly to the outside world.
    */
   private String number=new String();
   private int base;//2, 8, 10, 16, 754->IEEE754; custom bases allowed: 2 -> 16 for integers
   private boolean signed;//false->unsigned, true->signed (IEEE754 values must be signed)
   private int size;//# of bits: 8, 16, 24, 32, 64; custom sizes allowed: 1 -> 64 for integers

   //CONSTRUCTORS
   public NumericString()
   {
      //Set default values of the variable data fields
      this.number="00000000";
      this.base=BINARY;
      this.signed=true;
      this.size=BYTE_SIZE;
   }

   public NumericString(String number, int base, boolean signed, int size)
   {
      this.number=number;
      this.base=base;
      this.signed=signed;
      this.size=size;
   }

   public NumericString(NumericString n)
   {
      this.number=n.number;
      this.base=n.base;
      this.signed=n.signed;
      this.size=n.size;
   }

   //INSTANCE METHODS
   public byte byteValue()
   {
      //Return the "byte" value of this NumericString object
   }

   public char charValue()
   {
      //Return the "char" value of this NumericString object
   }
```

```
   public short shortValue()
   {
      //Return the "short" value of this NumericString object
   }
   public int intValue()
   {
      //Return the "int" value of this NumericString object
   }
   public long longValue()
   {
      //Return the "long" value of this NumericString object
   }
   public float floatValue()
   {
      //Return the "float" value of this NumericString object
   }
   public double doubleValue()
   {
      //Return the "double" value of this NumericString object
   }
   public int rgbValue()
   {
      //Return the rgb colour value of this NumericString object
   }
   public long unsignedValue()
   {
      //Return the unsigned integer value of this NumericString object
   }
   public long signedValue()
   {
      //Return the signed integer value of this NumericString object
   }
   public NumericString convertTo(int base, boolean signed, int size)
   {
      //Return this NumericString object converted to the given base
   }
   public NumericString convertTo(NumericString n)
   {
      //Return this NumericString object converted to the format of the NumericString object 'n'
   }
   public NumericString resize(NumericString n, int size)
   {
      //Return this NumericString object converted to the format of the NumericString object 'n'
   }
   //Static Methods → To be determined by the individual student.

}//end of class
```