Computer Science 202
Introduction to Programming
The College of Saint Rose
Fall 2013

# Topic Notes: Working with Numbers

We have already seen that computers often need to compute with numbers. In fact, when it comes down to it, they do nothing but compute with numbers. So next, we consider some examples of programs that work with numbers that don't have anything to do with graphical objects.

---

## Integer Values

We start simple. Let's compute a rectangle's area and perimeter.

See Example: Rectangle

There are a few things to note in this program.

First, we are working with numbers rather than words. This changes how we read the data from the keyboard through our `Scanner` and the type of variable we need to declare to store that data.

For this example, we are requiring that the dimensions of the rectangle are integer values.

As you know, the Java type we use to store an integer value is called an `int`. We declare and initialize `int` variables named `width` and `height` to store the rectangle's dimensions.

We next need to use a different method of `Scanner` to force it to look for an integer and return it in as a Java `int` instead of a `String`. That method is called `nextInt`.

Once we have our width and height, we need to compute the area and perimeter from them. For this, we need to declare two more `int` variables and perform some computation to compute their values.

If you remember your elementary school geometry, you know that to compute the area of a rectangle, we multiply its width by its height. And to compute the perimeter we add up the lengths of all sides, which in this case is twice the width plus twice the height.

Java uses a notation to specify mathematical computations (a mathematical *expression*) that is mostly familiar from math. As we can see from the statement that computes `area`, we use the `*` operator to specify multiplication.

So that statement instructs Java to multiply together the `int` value found in the variable `width` by the `int` value found in the variable `height` and store the product in the `int` variable `area`.

The computation of `perimeter` is a bit more complicated, but still pretty straightforward. We see that addition is specified by `+` and that we can use numbers in our expressions as well as values stored in variables.

We do need to know in what order Java will perform the operations here. If it does `2 * width`, then adds 2 to that result, multiplying that resuly by `height`, we will get the wrong answer.

Fortunately, Java follows a strict *order of operations*. In this case, we say that multiplication has a higher *precedence* than addition, so Java will compute `2 * width`, then `2 * height`, then add together those results.

We will look in more detail at order of operations as we encounter other mathematical operators in subsequent examples.

Finally, we print out our results. We can see here that Java "does the right thing" when we concatenate string literals with `int` values.

Question: what happens if we type in something that's not a valid `int`?

---

## Floating-point Values

Our next example is to perform a simple miles per gallon computation, which you will develop in class.

When we divide two `int` values using `/`, the result is the *quotient*, and we throw away the remainder. If we want the remainder (and only the remainder), we can use the `%` operator, often called the "mod" operator as it performs modulo arithmetic.

Any division operator where both operands are `int` values, results in an `int` quotient. If *either* operand (or both) is already a `double`, the results is a `double` and the answer would include any fractional part as a decimal.

---

## Operator Precedence

We can specify complex arithmetic expressions using any combination of the following:

| | |
|---|---|
| `*` | multiplication |
| `/` | division |
| `%` | remainder |
| `+` | addition |
| `-` | subtraction |

In a long expression such as

```
12 + 9 / 4 - 18 % 4 * 19
```

there are choices to be made in how to evalulate. Fortunately, Java makes these decisions and makes it clear to us how it will evaluate such an expression.

1. *unary* negation operators are applied first, working left to right if there are multiple such operations
2. multiplications, divisions, and remainders are computed, again left to right

3. additions and subtractions are computed, left to right

So in the above expression, we first check for unary negations, and there are none.

Then, we do the multiplication, division, and remainder operations. Since these are all integer values, the any division will be computed as an integral quotient.

So, the `9 / 4` evaulates to `2` first. Giving

```
12 + 2 - 18 % 4 * 19
```

Next, `18 % 4` is evaluated to `2` (the remainder when we divide 18 by 4). Giving:

```
12 + 2 - 2 * 19
```

One multiplication remains, so we compute the `2 * 19` as `38`, giving:

```
12 + 2 - 38
```

We are left with only additions and subtractions, which are evaulated left to right. `12 + 2` becomes `14`, leaving us:

```
14 - 38
```

and after the last subtraction, we have `-24` for a final result.

The same rules apply if we have data in variables declared as either `int` or `double` values.

If we wish to override the default rules, just like in math, we can place parentheses around any lower-precedence operation that we wish to have performed before some higher-precedence operation, or if we want to change the order among same-precedence operations to do some further to the right before some further to the left.

---

## Logical Operations Example

As a further example of a Java application and a way to look at many of the uses of the logical operators we studied recently, consider the following:

See Example: BooleanDemo

See the comments therein to see some details.

In particular, note the precedence of these operators: && is evaluated before `||`, much like multiplication is evaluated before addition in an arithmetic expression.

Important note: you need to be very careful that you do specify these operators as && and `||` rather than & and `|`. The single-character operators will perform a bitwise and (or) rather than a logical and (or), which is not usually what you want..

The only other new item here in terms of a Java construct is the ability to read in numeric data from a `Scanner`. We read in 3 `int` values to our program by calling the `Scanner`'s `nextInt` method. We can then use those numbers in a series of comparisons and other logical operations.

---

## Numeric Data Types

A few more words are needed about numeric data.

Java includes four commonly used numeric types:

- `int`,
- `long`,
- `float`
- `double`

(There are others, but we won't discuss them here.)

The types `int` and `long` both represent integers. Type `int` represents numbers between about $-2 * 10^9$ and $2 * 10^9$, while `long` represents integers up to about $10^{19}$.

Numbers written with decimal points are represented by the types `float` and `double`. The type `float` only retains about 7 digits of precision, so we usually use `double` instead. Type `double` has about 15 digits of precision and can represent numbers up to about $10^{308}$ and as small as $10^{-308}$. These numbers can be written either simply with a decimal point, e.g., $4.735$, or in scientific notation, *e.g.*, $5.146$E+47, representing $5.146 * 10^{47}$.

Because of the way many of the Java libraries are written we will tend to use type `int` for integers and `double` for numbers with decimal points. Occasionally we will use `long` when we need to represent more digits with integers, but we will find no need to use `float` in this course.

Like `int`, the type `double` supports operations +, -, *, and /, but does not have an operation corresponding to %. The results of applying these operators to `double`s is an answer which is also a `double`. Thus 3.0 / 2.0 = 1.5.

If an arithmetic operator has one operand with type `int` and the other with type `double`, the result will be a `double`. Basically what happens is that Java recognizes that the two operands are of different types (which it is not happy about), and thus attempts to make them be of the same type. The simplest thing, from Java's point of view, is to convert `int`s to `double`s, as no loss of precision results. As we saw earlier, if both operands are of type `int`, the result will also be of type `int`.

One must be careful in performing divisions to be aware of the types of the operands, as the results may differ depending on their types. *e.g.*, 3 / 2 = 1, while 3.0 / 2.0 = 3.0 / 2 = 3 / 2.0 = 1.5.

## Representing Data and Information

Before we go any further, it is necessary to think about how we can represent data and information for use by a computer. We think of our computers dealing with numbers, text, pictures, sounds, videos, and more. All of these things must be encoded in a way that a computer can gather, process, output, and store it.

The fundamental idea here is that as a computer is really just a collection of electric circuits. Those circuits transmit and store electrical signals that are either off or on. Those signals encode the only two values a computer understands: the values 0 and 1.

Anything more complicated that we wish to use must be encoded using a sequence of these 0's and 1's, which are known as *bit*s, short for **b***inary dig***its**.

The function of any computer can be boiled down to this: it takes a collection of bits, some of which are data and some of which are instructions on what to do to that data, and performs those instructions, producing a new collection of bits.

### Binary Basics

Question: how high can you count on one finger?

That finger can either be up or down, so you can count 0, 1 and that's it.

(Computer scientists always start counting at 0, so you should get used to that...)

So then... How high can you count on one hand/five fingers?

When kids count on their fingers, they can get up to 5. The number is represented by the number of fingers they have up.

But we have multiple ways to represent some of the numbers this way: 1 0, 5 1's, 10 2's, 10 3's, 5 4's and 1 5.

We can do better. We have 32 different combinations of fingers up or down, so we can use them to represent 32 different numbers.

Given that, how high can you count on ten fingers?

To make this work, we need to figure out which patterns of fingers up and down correspond to which numbers.

To keep things manageable, we'll assume we're working with 4 digits (hey, aren't fingers called digits too?) each of which can be a 0 or a 1. We should be able to represent 16 numbers. As computer scientists, we'll represent numbers from 0 to 15.

Our 16 patterns are base 2, or *binary*, numbers. Again, we call the digits *bits*.

Each bit may be 0 or 1. That's all we have.

Just like in base 10 (or *decimal*), where we have the 1's ($10^0$) place, the 10's ($10^1$) place, the 100's

($10^2$) place, etc, here we have the 1's ($2^0$), 2's ($2^1$), 4's ($2^2$), 8's ($2^3$), etc.

As you might imagine, binary representations require a lot of bits as we start to represent larger values. Since we will often find it convenient to think of our binary values in 4-bit chunks, we will also tend to use base 16 (or *hexadecimal*).

Since we don't have enough numbers to represent the 16 unique digits required, we use the numbers 0–9 for the values 0–9 but then the letters A–F to represent the values 10–15.

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

Any number can be used as the base, but the common ones are base 2, base 8 (*octal*, 3 binary digits), base 10, and base 16.

A byte (8 bits) of data can be written as a decimal number in the range 0 to 255, as 8 binary bits, or as two hexadecimal symbols.

Two bytes of data: decimal in the range 0-65535, 16 bits, or 4 hex digits.

---

## Decimal-Binary-Hex Conversions

Since we will encounter values in decimal, binary, and hexadecimal at various times, we should think about how to convert among these representations.

We'll first look at the easiest case: converting back and forth between binary and hex. Each hex digit represents 4 bits and each group of 4 bits can be represented by a hex digit.

So to convert the hex value `7A4D` to binary, we convert each digit:

```
0111 1010 0100 1101
```

And to go back the other way, we can start with

```
0100 1011 0000 0110
```

(conveniently written in groups of 4 bits) to obtain `4B06` in hex.

Converting from binary or hex to decimal is also pretty straightforward. We just add up the values represented by each digit. Start with a binary value:

```
01011011
```

This binary string is interpreted:

$$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

which is 64+16+8+2+1 = 91 in decimal.

Notice that this is **exactly** the same way we interpret a decimal value. The value 2872 in decimal is really:

$$2 \times 10^3 + 8 \times 10^2 + 7 \times 10^2 + 2 \times 10^0$$

It's a similar idea converting from hex. Here, we'll look at the place values in base 16. Starting with the hex value `30C7`:

$$3 \times 16^3 + 0 \times 16^2 + 12 \times 16^1 + 7 \times 16^0$$

We'll probably grab a calculator for this:

$$3 \times 4096 + 12 \times 16 + 7 = 12288 + 192 + 7 = 12487$$

The remaining conversions start from decimal. We'll first convert decimal to binary. We assume here that we are going to store our result in 8 bits, but the procedure can be extended to larger values.

We will use 108 (decimal) as our value.

Let's remember what each position in our 8-bit binary value represents:

The first bit is the number of $2^7 = 128$'s in the number.
The second bit is the number of $2^6 = 64$'s
The third bit is the number of $2^5 = 32$'s
The fourth bit is the number of $2^4 = 16$'s
The fifth bit is the number of $2^3 = 8$'s
The sixth bit is the number of $2^2 = 4$'s
The seventh bit is the number of $2^1 = 2$'s

The eighth bit is the number of $2^0 = 1$'s

We work in order from left to right. If the number is greater than or equal to the value stored in a position, we place a 1 in that position and subtract the value for that position from the number. Otherwise, we place a 0 in that position.

So for our number 108, we start with the 128's place. 108 is smaller, so we place a 0 there:

```
0???????
```

Next, we notice that 108 is larger than 64, so we place a 1 in the 64's place and subtract 64 from our number. Subsequent steps will work with the number 44.

```
01??????
```

44 is greater than 32, so we have a 1 in the 32's place, and are left with 12 to work with.

```
011?????
```

There are no 16's in 12, so we put a 0.

```
0110????
```

There is an 8 in 12, so we place a 1 and subtract, leaving us with 4.

```
01101???
```

And there is a 4, so we place a 1 in the 4's and are left with 0.

```
011011??
```

The 0 remaining will not contain any 2's or 1's, so we will be placing 0's in the final two positions.

```
01101100
```

And there's our answer.

We will follow a similar procedure to this to convert decimal to hex, but it's a little more tricky since we're dealing with powers of 16.

Here, we will assume that we want the answer in 4 hex digits, and we'll start with the decimal number 19,832.

First, we need to remind ourselves what the place values are in hex:

$16^3 = 4096$
$16^2 = 256$
$16^1 = 16$
$16^0 = 1$

So we begin by figuring out how many 4096's there are in 19,832. If we divide 19,832 by 4096, we get 4, with a remainder of 3448. So our first hex digit will be a 4.

```
4???
```

And we continue working with that remainder, 3448. Note that you can also think about subtracting $4 \times 4096 = 16384$ from our starting number to get that remainder.

How many 256's are there in 3448? Well, $\frac{3448}{256}$ gives us 13, with a remainder of 120. This means we need to use 13 as our second hex digit. Recall that the character we use to represent 13 is D.

```
4D??
```

Continuing on, we are left with 120 and we need to fill in the 16's place. $\frac{120}{16}$ gives 7 with a remainder of 8. So we have our last two digits, and the hex representation of our number is

```
4D78
```

---

## It's All Binary

So we have seen how we can represent unsigned (*i.e.*, non-negative) whole numbers in binary. However, that is just one of many kinds of data that we will wish to store and process.

- Integers (including negative numbers): the representation is similar to what we use for unsigned numbers. The usual representation is called *2's Complement*, but the details are not our concern.
- "Real" numbers with fractions/decimal points: there are many choices of how to represent non-intergral values. The representations almost always used are called *floating point* representations. Not all values can be represented exactly – approximations are needed.
- Characters: representing text. The two most common mappings of binary values to characters:
  - The *American Standard Code for Information Interchange* (ASCII) – provides a set of one-byte representations for English characters, numerical digits, and common punctuation.

    See: `http://en.wikipedia.org/wiki/ASCII`

- *Unicode* – introduced about 20 years ago – representations are 2 bytes per character, allowing the inclusion of many international characters.

  See: `http://en.wikipedia.org/wiki/List_of_Unicode_characters`

  Text such as that in the document you are reading now is represented as a sequence of ASCII or Unicode characters.

- Other media, such as sound, images, and video each have many possible representations. When we refer to an "MP3" audio file, or a "JPEG" image, or an "MPEG" movie, we refer to a file that follows a specific set of rules about what bit patterns correspond to the sound, image, or video being represented.
- Computer instructions: both the data and instructions are stored as binary values in the computer's memory. We will look at this briefly in the next section.

---

## Units for Measuring Data

While the fundamental unit of information is the bit, the smallest unit of data usually considered is a group of 8 bits, called a *byte*. A byte is enough to store any one of 256 (=$2^8$) values. Typically, a byte can represent a number or a letter or a special character (like a `!`). We will look more carefully later in the semester at how bytes can be used to represent specific number or characters.

To represent anything beyond the simplest data, we will need to use lots and lots of bytes. A page of text requires a few thousand bytes. So a several hundred-page book requires over a million bytes.

This leads us to some prefixes to indicate numbers, many of which you have likely heard used in this context:

| Unit | Abbrv. | Size | |
|---|---|---|---|
| Kilobyte | KB | $2^{10} = 1024$ bytes | 2.4% more than $10^3$ (thousand) |
| Megabyte | MB | $2^{20}$ bytes | 4.8% more than $10^6$ (million) |
| Gigabyte | GB | $2^{30}$ bytes | 7.3% more than $10^9$ (billion) |
| Terabyte | TB | $2^{40}$ bytes | 9.9% more than $10^{12}$ (trillion) |
| Petabyte | PB | $2^{50}$ bytes | 12.5% more than $10^{15}$ (quadrillion) |
| Exabyte | EB | $2^{60}$ bytes | 15.2% more than $10^{18}$ (quintillion) |
| Zettabyte | ZB | $2^{70}$ bytes | 18% more than $10^{21}$ (sextillion) |
| Yottabyte | YB | $2^{80}$ bytes | 21% more than $10^{24}$ (septillion) |

See: `http://www.unc.edu/~rowlett/units/large.html`

Note that each entry in the table is 1000 times larger than the previous. These are some incredibly huge numbers!

These prefixes are used elsewhere in describing sizes in computer technology. A camera's "megapixels" indicates how many millions of pixels are in the images that the camera can capture.