Section 1 - Numeric Types

2B.1.1 Integer Types

You have already seen the **int** data type. When we explored the exciting statements

```
   int someNumber;

   someNumber = -7;
   someNumber = (someNumber - 2) / 3;
```

we were working with **int** variables and **int** values. An **int** variable can hold a whole number in the range -2,147,483,648 to +2,147,483,647.

However, if we want a larger range of integers, we can use the **long** data type:

```
   long s;
```

Now **s** can hold a larger integer.   Of course, a **long** variable is still capable of working on the usual integers, but it can go higher (or lower) if we need it to. **long** constants are distinguished from **int** constants by the letter **L** (or **l**) after the number. For example, using our long variable, **s**, declared above we can make the following assignments:

```
   s = 123456789123456789L;
   s = -1L;
```

The latter assignment statement could have been written without the L, because the compiler would take an ordinary -1 and convert it to the longer format for you.

You can also force an integer to be *shorter* than usual by using the **short** data type:

```
   short s;
```

In this case, **s** can only hold integers in the range -32,768 to 32,767. The only reason to use shorter integers is to save memory. If you have an array of billions of integers, you would want each one to be as short as possible (we will get to arrays later).

So there are three different primitive types, **int**, **long** and **short**, which are all members of the integer data type family.

When sending any integer type to the screen through a **println() or showMessageDialog()** method, you can forget about what type they are and just insert them into the output stream using the concatenation operator:

```
   short  shrt;
   long   pants;

   shrt = -1;
   pants = 123456789123456789L;

   System.out.println( "shrt is " + shrt
         + " and pants is " + pants  );

/* ------------ paste of output -------------

shrt is -1 and pants is 123456789123456789

------------------------------- */
```

Section 2 - Floating Point Types

2B.2.1 Float and Double Types

Next, we visit the floating point data types, **float** and **double**. These are each capable of holding numbers that include decimals and fractional parts. **float *constants***, or ***literals***, are distinguished by writing the letter **F** or **f** after the constant, as in:

3.14159f

or

5.7e+9F

double constants have no letters after them. So if you type a number like **123.456** you are implying that it is a **double** not a **float**.

double gives considerably more precision. If you need 8 or 9 decimal places of accuracy, **float** will do:

100.3032f
1.23232e-14f

but for 10 or more decimal places I use **double**:

100.3032321100194
-.0000000000001
2.1231234408456345e-5

Remember, constants with a decimal point but without any special character after them are assumed to be **double**. With an "f" following them, they are assumed to be **float**:

```
double s;
s = 1.23; /* even if you don't use the precision
             it's there */
float r;
r = 5.4f;
```

You should probably always use **doubles** -- they are easier to write and more powerful. If you see me talk about or use a **float** in this class, you can usually just replace it with a **double**.

Java takes care of displaying each object using the proper notation without messy formatting required by us programmers:

```
float  f;
double  l;

f = 31.23232e-14f;
l = 123456789123.456789;

System.out.println( "The numbers are  " + f + " " + l  );
```

results in:

```
The numbers are  3.123232E-13 1.2345678912345679E11
```

Note that I placed `" "` (a string literal consisting of just one space) between the numbers. That makes it possible to see them without confusion. If, instead, we had tried this without the space like so:

```
   System.out.println( "The numbers are  " + f + l  );
```

our output would have been:

```
The numbers are  3.123232E-131.2345678912345679E11
```

NOTE: Here the + sign means concatenation, not addition, even between the two numeric types f and l. This might seem odd. We will see why shortly.

Section 3 - Char Types

2B.3.1 char Type

We have seen the **String** class which is used to hold character strings with mixed letters and numerals in them. However, sometimes we only need a single character. There is a special primitive data type for alpha-numeric data like the letters, numbers and symbols on the condition that you only want to store one character in each variable. This type is **char**.

A **char** object can hold a single alphanumeric character, like this:

```
   char letter;
   letter = 's';

   System.out.println( "The letter is  " + letter  );
```

results in:

The letter is s

Notice that the character **'s'** is contained within single quotes. This means that the computer is to store a 16-bit (two byte) ***Unicode*** code representing the letter **'s'** in the **char** object **letter**. These codes are universal and can be found in many references, but you really don't need to know them, just that they exist.

The first 127 (printable) of these codes are the same as the ***ASCII codes*** for chars used in C++. Because of this, I will often say **ASCII** when referring to ordinary char data of printable characters.

Note that the statements

```
   letter = '3';
```

and

```
   x = 3;  // assumes x is an int
```

are very different. The value stored in **x** is an actual numeric **3**, whereas the value stored in **letter** is the **ASCII** (Unicode) code for the numeral **'3'** (which happens to be a number *other* than **3**).

The **char** type can hold non-printable codes as well. The ***tab*** and ***newline*** keys have codes that we can use. For these special characters, we use notation like:

```
   '\t'  tab
   '\n'  newline
   '\0'  null or zero
```

Even though there are two characters between the single quotes, a slash plus an extra one, the value of the constant is still a single ***Unicode*** character. The last example above shows the constant for the char value = 0. This is different from the **ASCII** code for **'0'** whose value is *not* **0**. Thus, **'\0'** stands for the byte with all bits turned off (set to 0).

Section 4 - Strings

2B.4.1 Declaring and Initializing Strings

We've seen **Strings** briefly in the form of ***String literals***. These were phrases inside quotes that we wanted to display in output statements like:

```
   System.out.println( "someNumber is " + someNumber );
```

The phrase "someNumber is" is a ***String literal***, and **someNumber** is an **int** variable that holds a numeric value. But can we have a **String** ***variable*** that holds a **String** ***value***? You bet. We declare a **String** ***variable*** just as we would an **int**, except we use the word **String** instead of **int**:

```
   String thxMom;
```

So far, so good. But what about assigning a ***value*** to this **String**, **thxMom**? This part is a little stickier because **Strings**, unlike **ints**, are not primitive data types. The data type **String** is actually a **class**, a kind of data type that is more powerful and flexible than a primitive type like **int**.

What this means is that we build up a **String** ***object*** using a special notation. It is as follows:

```
  thxMom = new String("Thanks, Mom!");
```

Once we have done that, we can use the variable **thxMom** as a String variable that contains the value "Thanks, Mom" stored in it. Often we put the two statements near each other, as in:

```
   String thxMom;                          // declare the reference
   thxMom = new String("Thanks, Mom!");   // create the object
```

Let's break down these two statements so you understand them. Also, we'll introduce some very important vocabulary.

The first statement declares a variable, **thxMom**, of the ***class*** **String**. **thxMom** is also called a **String *reference*** or ***pointer***. It is capable of referring to or pointing to a **String**, but it doesn't do either of those things until the second statement is reached.

The second statement creates a **String *object*** that holds the **String *value*** "Thanks, Mom!" and lets the variable **thxMom** point to, or ***reference***, that new object.

It seems like a lot of work to establish a simple **String** object, but believe me, it's worth it. These two statements give us incredible power to do things (most of which we won't learn for a few weeks).

2B.4.2 Using Strings in Programs.

It is incredibly easy to use **Strings**. Here is a simple ***console app*** example:

```
public class Experiment_2
{
   public static void main(String[] args)
   {
      String str;
      str = new String("Hello, there - Java is easy!");

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

This, as you can imagine, sends the following to the console:

```
Hello, there - Java is easy!
```

I realize that we could have created the same effect with the single statement:

```
   System.out.println("Hello, there - Java is easy!");
```

However, declaring a **String *variable*** instead of using a **String *literal***, has some advantages as we shall see.

2B.4.3 Class, Reference and Object

What we have just seen is a perfect example of the concepts of ***class***, ***reference*** and ***object***.   We will use these terms throughout the latter part of the course.

Let's say we wish to use a class named **String** to help us with our program. In order to work with specific **String** data, we first have to ***declare a reference*** to the class:

```
   String str;
```

In case it isn't obvious, we could have named our reference something besides str. We could have called it **thxMom**, or **output** or just **x**.

At some point, before we can work with **String** data, we have to ***instantiate an object of the class*** and ***point to it*** with our ***reference***. All this is done in the next statement:

```
   str = new String("Hello, there - Java is easy!");
```

Once these two steps are complete, we can start using the String str. It has data that we can play with or display on screen.

In the future, we may end up using a class, **Automobile**. We might want to create an **Automobile *reference*** called **myCar**, and then ***instantiate*** an object of that class for **myCar** to point to. It might look something like this:

```
// declare Automobile reference
Automobile myCar;

// instantiate Automobile object for myCar
myCar = new Automobile("Toyota Prius");
```

2B.4.4 A Simpler Notation For Strings

I went out of my way to accentuate the difference between **String** data and primitive numeric data. I made it seem like **String** data required a special instantiation step in addition to the declaration step. This is only half true. Here's the other half.

You can actually work with **String** references much like you work with primitive variables. In other words, we can declare and initialize a **String** reference like this:

```
String myName;
myName = "Wayne Palmer";
```

This looks a lot more like the primitive data types **int** or **float** because we don't have any nasty syntax like myName = new String("Wayne Palmer"). This shorter notation is for convenience only. Unlike primitive numeric types, even with this simpler notation we are not storing **"Wayne Palmer"** in location **myName**. **myName** is still a String ***reference***, and we are pointing it *to* an object that contains the ***literal*** **"Wayne Palmer"**. You can use this shortened notation, as I will, as long as you remember that **myName** does not *contain* the String value but merely *points to it.* This will be a useful fact.

The bottom line is this: you can make assignments to **String** references as though they are simple variables.

Section 5 - Converting Between Types

2B.5.1 Compatibility *Within* the Two Numeric Types

Generally, all ***integer types*** (**short**, **int**, **long**) can be mixed, as can both ***float types*** (**float**, **double**). In other words, you can assign one **int** type to another or one **float** type to another.

Note 1) You can put a smaller (lower) type into a larger (upper) type without any fanfare:

```
short s;
long l;

s = 3;
l = s;          // no cast needed to convert up, or "widen"
```

This is called "data widening."

Note 2) You must add extra notation if you wish to convert a larger type into a smaller one:

```
    s = (short)l;   // cast needed to convert down or "narrow"
```

This is called "data narrowing."

This extra notation -- placing the target data type in front of the value on the ***RHS***. (right hand side) of the expression -- is called a ***cast***. You are ***casting*** the **long l** to a **short s**.

Casting reminds you that you may lose data precision because you are throwing away accuracy. Since putting a **float *value*** into a **double *variable*** doesn't cost accuracy, there is no need to do anything special.

Terminology:

Sometimes **casting** is called **coercion** or **type coercion**.

Here is an example:

```
   float x;
   double y, ans;
   short s;
   long l;

   y = 1.0;
   x = (float)y;    // cast needed to convert "downward"
   y = x;           // no cast needed to convert up

   l = 123;
   s = (short)l;    // cast needed to convert "downward"
   l = s;           // no cast needed to convert up

   l = (long)y;      // cast needed to convert "downward"
   x = l;
```

Reminder:

You will lose points if you use a single letter variable name in most situations.

You see me using single letter names here because I am trying not to confuse you with words in these early examples. However, in actual practice and in your assignments you would use names like **year** instead of **y**, or **age** instead of **x**. The name of the variable should reflect what role it plays in your program.

2B.5.2 Compatibility *Between* the Two Numeric Types

There is a certain amount of compatibility between ***floating point*** types and ***integer*** types. In expressions like

```
int n;
float x;
double y;

n = 1;
x = 3.2F;

y = (2 + (x + n)) / (1.5 - n);
```

whenever a single binary operation has two numeric types to be mixed, the compiler ***promotes*** the lower type to the higher type: **int** gets promoted (temporarily) to a **float**, or a **float** could get promoted to a **double**.

That's half the story. In the assignment statement:

```
int m;

m = (2 + (x + n)) / (1.5 - n);
```

the expression on the right evaluates to a float because of ***promotion***, but then it has to get stored in an **int** variable, **m**. A **float** won't fit into an **int**, so we have to *order* the compiler to squeeze it in, anyway, using the syntax called type ***coercion*** (you should know the English verb "coerce" well enough to understand why it an appropriate choice here).

```
m = (int) (  (2 + (x + n)) / (1.5 - n)  );
```

So the fractional part is tossed out and the number is placed into **m** as an **int**.

If it had been

```
   x = n + 5;
```

then the right hand side of the assignment would evaluate to an **int**, after which it would be automatically converted to a **float** for storage into **x.**

What do you think would happen in this case (assuming **n** to be an **int** and **x** to be a **float**)?

```
   n = 5;
   x = n / 2;
```

2B.5.3 Ints and Chars

chars are just short ***integers*** -- they are integers that have only two bytes for their storage. Even though we introduced chars as **ASCII** or ***Unicode*** codes for printable characters, we could, in fact, consider **chars** as small **ints**:

```
   char p;

   p = 65;
   System.out.println( "The answer is  " + p  );
```

As long as the number is between 0 and 64000 (approx), it can fit into a **char**. However, regardless of how we establish the assignment, through an integer, or through a character, when printed out, Java will show it as a character:

```
   char p;

   p = 65;
   System.out.println( "The answer is  " + p  );

   p = 'c';
   System.out.println( "The answer is  " + p  );
```

producing output:

```
The answer is  A
The answer is  c
```

Just as **chars** always print out as letters, so **ints** always print out as values. You can use this as a way to print out some **ASCII** values, as we show in the next section. (Since **ASCII** and ***Unicode*** are equivalent for the standard character set, we will use these two terms interchangeably).

```
   int n;

   n = 'A';
   System.out.println( "The answer is  " + n  );
```

producing:

```
The answer is  65
```

So, when I speak of an ***integer data type,*** I include the **char** type since it is really closer in form to an **int** than it is to a **String**.

2B.5.4 Converting Between Strings and Numbers

There is a vast difference between:

```
   String s;
   s = "-3.4";
```

and

```
   double c;
   c = -3.4;
```

You tell me what it is.

What you hopefully said is that, while **s** will look the same as **c** when printed, **s** is incapable of being used in any computation. **s** is not a number, it is a **String**. You cannot compute directly using **Strings**. Even if a **String** happens to hold a numeric set of characters, this is unknowable to Java.

We usually get input from the user in the form of a **String**, not a number, by designing our program to use a **String** variable, such as **s**, above, to capture the user's answer. (I haven't told you how to read values from the user yet, so just go with me on this for now.) Imagine, then, that our program asks the user for a number, he types **-3.4** , and we store it in the **String s** . What we have is a **String "-3.4"** stored in **s**. Our next order of business is to turn that into a **double** -3.4, which will be held in our double variable, **c**, but how?

We use the notation:

```
   c = Double.parseDouble(s);  // convert String s to double c
```

We would do this, for instance, if we read a user response into the **String** object **s** and needed to convert it to a number in order to compute with it.

This can be duplicated with **int**, **float**, etc. Here's the int version:

```
   int ageAsInt = Integer.parseInt(ageAsString);
```

The Other Direction

The other direction is easier. If you want to change *any* primitive type to **String**, you do it like so:

```
s = String.valueOf(c);
```

where c can be **double, float, int,** etc. It can even be an expression:

```
s = String.valueOf( 3.7/(x+y) ); // x, y assumed double
```

I think you can see by comparison to the above, that this time the **String *class*** is being used to invoke a ***method*** called **valueOf()**. The ***method*** is a ***function*** that takes a number as input and ***returns*** a **String** object.

However, I have to point out that this is usually not necessary. You may not have noticed it, but you have seen this trick to convert any numeric type to a **String**. I showed you how to combine **Strings** and ***numbers*** in your **println()** statements by using the + concatenation operator. We can do that anywhere, not just inside println() statements:

```
s = "" + (3.7/(x+y));
```

The idea is that we start with an ***empty*** **String**, **""** (nothing between the double quotes) and concatenate (with +) any number or numeric expression we wish.

2B.5.5 Converting Between Strings and Chars

A **String** object can hold many characters, but a **char** can only hold one:

```
String myString;
char myChar;

myString = "Hi Mom. Hi Alan";
myChar = 'Q';
```

It is very common to want to extract a character from a **String** and place it into its own **char** variable. That's done using the **String** method **charAt()**, as follows:

```
String myString;
char myChar;

myString = "Hi Mom. Hi Alan";
myChar = myString.charAt(3);
```

We always begin counting from *zero* (**0**) in programmings, so the third p0sition of **myString** holds the letter **'M'**. Thus, **myString.charAt(3)** extracts the **'M'** from the string **"Hi Mom. Hi Alan"**. It then stores that character into **myChar**.

Question for Discussion:

How would you get the first character from a **String** object, **userName**, and print it to the screen, immediatly, *without* using a **char** variable? How would you first store it into a **char** variable then print it to the screen using that char variable?

Section 6 - String Concatenation

2B.6.1 The Concatenation Operator

We've seen the ***concatenation operator, +,*** in action before:

```
   System.out.println( "someNumber is " + someNumber );
```

The ***concatenation operator*** is used to join together two or more **Strings**. We can ***concatenate*** either **String variables** *or* **String literals**.

The next program will help you if you ever receive an Oscar at the Academy Awards. It will demonstrate use of the ***concatenation operator, +***, as well as the new **String** data type that we just presented. We will create a speech by piecing together certain phrases.

Notice how we first declare the **String *references***, then later use them to ***instantiate*** specific **String *objects*** with phrases in them. Also, there is one new item that you'll see: an example where we combine the **String** declaration with the object instantiation into one statement.

Have a look:

```
public class Foothill
{
   public static void main(String[] args)
   {
      // declare some string references (variables)
      String thxMom, thxAgent, thxFox;
      String outputString;

      // for fun, create the String object directly in the declaration
      String acceptanceSpeech = new String("I'd like to thank ");

      // create the rest of the  string objects to use
      // in your speech.
      thxMom = new String("my Mother, Reva, and wife Coleen.");
      thxFox = new String("everyone at Fox and FBC.");
      thxAgent = new String("my agent and everyone at Paradigm.");

      // stand up at the podium and get settled ...
      // for this use String Literals directly in the
      // output statements.
      System.out.println(
         "I didn't really expect to win ...");
      System.out.println(
         "I don't even have a speech prepared!");
      System.out.println( "Anyway ...");

      // now finally start to thank people.
      outputString = "First of all " + acceptanceSpeech + thxFox;
      System.out.println( outputString);

      outputString = "Next, " + acceptanceSpeech + thxAgent;
      System.out.println( outputString);

      outputString = "But mostly, " + acceptanceSpeech + thxMom;
      System.out.println( outputString);
   }
}
```

This program has a few variations of what we learned:

- There are several instances of the ***concatenation operator*** being used to glue together **String *literals*** and **String *variables*.**
- We have a short-cut statement that ***declares*** and ***instantiates*** a **String *object*** in one line. Can you find it?
- We used a **String *reference***, **outputString**, to hold the result of some **String *concatenations***. Do you see this?

Paste this into your compiler IDE and run it. It's a fun little program and you can use it as a basis for doing homework assignments.

Section 7 - Sample Programming Labs

Now that we understand a lot more about how to write Java programs, some improved assignment examples can be given. Here are a couple model homework submissions to remove any remaining confusion about how the homework should look.

2B.7.1 Typical Lab Assignment Submission #1

FAKE HOMEWORK #1:

Write a program to determine and print out the ASCII codes for the letters 'L' and 'l', 'F' and 'f'. What can you deduce about the uppercase and lowercase ASCII codes?

Here is how you would solve it, and this is what your homework submission should look like:

```
/*
 *  Source program for Fake Homework #1 for CS 1A
 *  Written by Michael Samuel, 11/15/2012
 *
 */

public class Foothill
{
   public static void main(String[] args)
   {
      char upperLet, lowerLet;
      int upperVal, lowerVal;

      // first do the L/l values. place in both char and int and show:
      upperLet = 'L';
      lowerLet = 'l';
      upperVal = upperLet;
      lowerVal = lowerLet;
      System.out.println( upperLet + " has ASCII value: " + upperVal );
      System.out.println( lowerLet + " has ASCII value: " + lowerVal + "\n"
);

      // next do the F/f values. place in both char and int and show:
      upperLet = 'F';
      lowerLet = 'f';
      upperVal = upperLet;
      lowerVal = lowerLet;
      System.out.println( upperLet + " has ASCII value: " + upperVal );
      System.out.println( lowerLet + " has ASCII value: " + lowerVal + "\n"
);
   }
}

/* ------------------- Sample Run --------------------

L has ASCII value: 76
l has ASCII value: 108

F has ASCII value: 70
f has ASCII value: 102

---------------------- End Sample Run ---------------- */
```

```
The conclusion I draw from this is that the lower case letters
have ASCII codes which are exactly 32 larger than their
upper case counterparts: i.e., 'a' = 'A' + 32 and 'z' = 'Z' + 32.
```

2B.7.2 Typical Lab Assignment Submission #2

FAKE HOMEWORK #2:

Compute the following three expressions for *x* = -3 and *y* = 10:

$$x^3 + y^2$$

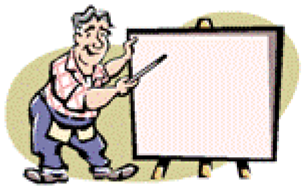
$$\frac{x - y}{x + y}$$

$$2 + 4 + \ldots + y$$

Here is how you would solve it, and this is what your homework submission should look like:

```
/*
 *  Source program for Fake Homework #2 for CS 1A
 *  Written by Nina Meyers, 1/1/2012
 *
 */

public class Foothill
{
   public static void main(String[] args)
   {
      float x, y, answer;

      // set up input values for variables x and y:
      x = -3;
      y = 10;

      // first expression:
      answer = x * x * x + y * y;
      System.out.println( x + "^3 + " + y + "^2 = " + answer + "\n" );

      // second expression
      answer = (x - y) / (x + y);
      System.out.println( "(" + x + "-" + y + ") / ("
         + x + "+" + y + ") = " + answer + "\n" );

      // third expression
      answer = 2 + 4 + 6 + 8 + 10;
      System.out.println( "2 + 4 + ... + " + y + " = " + answer + "\n" );
   }
}

/* ------------------- Sample Run --------------------

-3.0^3 + 10.0^2 = 73.0

(-3.0-10.0) / (-3.0+10.0) = -1.8571428

2 + 4 + ... + 10.0 = 30.0

---------------------- End Sample Run ---------------- */
```

Prevent Point Loss



- **Do not create lots of variables** when you can re-use the same variables. This is particularly true if the variables all represent the same kinds of things. For example, it would usually be better to have one variable, age, rather than age1, age2 and age3. However, don't overdo this principle - see next bullet. (1-2 point penalty)
- **Do use distinct variables** if they represent different sorts of things. For example, don't use the same variable for input as you use for output. (1 - 2 point penalty)
- **Describe output** in screen output statements. Undocumented, or under-documented values in your screen output are unacceptable. This documentation should be generated by your program, not added by hand to your output. (1 - 2 point penalty)
- **Do not do computations in a screen output statement.** In general, don't mix computation and input or output. (2 - 3 point penalty)