CS125 : Introduction to Computer Science

Lecture Notes #3
Types, Variables, and Expressions

Lecture 3 : Types, Variables, and Expressions

The "program skeleton"

During the last lecture, we saw this simple Java program:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

We're not going to go into any detail right now on what most of the above code means. It's not important (yet). So don't worry about trying to understand that code. However, you do want to have everything but the line `System.out.println("Hello World!");` in every program you write. That is, you want to make sure you have the following "program skeleton" when you write a program:

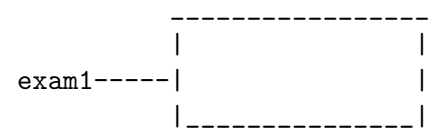
```
public class ProgramName
{
   public static void main(String[] args)
   {
         (the stuff you learn these next
            few weeks would go here)
   }
}
```

Basically, you can copy those six lines – four of them are curly braces – into your file, and then in between the inner set of curly braces, you can add the stuff we talk during the next few lectures. Until we start talking about *methods* in lecture 11, the only thing in the "program skeleton" above that will ever change is the `ProgramName`. The rest of it stays exactly the same, and all you will do is write different code within those inner curly braces.

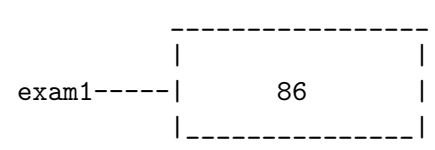
Variables

A *variable* is a name which refers to a value. You've seen this concept in math class before, where you would be given equations such as $y = 3x + 5$ and then various values could be substituted for x or y. It's the same sort of idea in computer programs, except that in addition to one-character names like `x` or `y`, our variable names can also be longer names such as `totalProfits`, `exam1`, `exam2`, or `numStudents`.

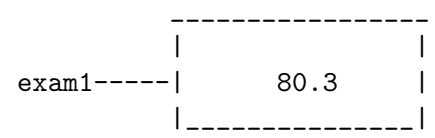
One useful way to think of a variable is to imagine it as a box with a label. If we have a variable named `exam1`, then think of it like a box labelled "exam1":



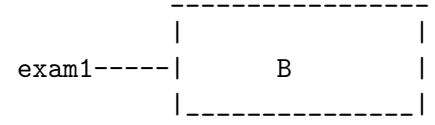
and then various values could be stored in that box, such as an integer:



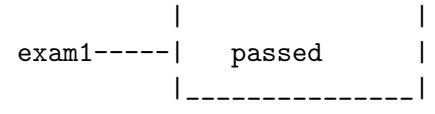
or a floating-point value:



or an individual character:



or an entire word:



A variable in a computer program can only hold one value at a time, but sometimes that variable is very simple (such as a single character) and sometimes it is more complex (such as a word, made up of many characters).

Types

One problem with the concept of variables is that the variable might hold a value that doesn't make sense, given how we are trying to use that variable. For example, if you were using the equation $y = 3x + 5$ in math class, and you were told that x was the word "hello", that would not make any sense! You are expecting x and y to stand for numbers, not words. If x is the word "hello", well, how on earth do you multiply "hello" by 3 and then add 5 to it? The idea is silly.

So, to prevent this sort of problem from happening, some computer languages require the programmer to associate a *type* with each variable. A *type* is the "kind" of data that variable is allowed to hold. The programmer will have to decide in advance, "this variable will hold only integers" or "this variable will hold only floating-point numbers" or "this variable will hold only words". The compiler is then responsible for making you stick to that decision. For example, if you decided a variable would hold only integers, and then tried to store a word in that variable, the compiler would complain that your variable type and your value type don't match (this is sometimes referred to as a "type mismatch"). Your program, therefore, would not correctly compile until you fixed this problem.

Java is such a language. In Java, every variable has a particular type, and you must decide what that type will be before you can use the variable in the rest of your program. So, before we go any further, we need to learn what types of data we can have our variables store. The types you are about to learn are specific to Java, but other languages have their own types that might be similar or identical, or might be different.

There are 8 essential types in all...called the 8 "primitive" types. (Later in the semester, we will design our own, more complicated types, but those types will use the eight primitive types as building blocks.)

Integral types

- Four types: `byte`, `short`, `int`, and `long`
- These four types all refer to integers, but have different size restrictions.
- A `byte` takes up 8 bits of memory, and as a result can be one of exactly 256 different values. ($2^8 = 256$.) The actual range is -128 through 127.
- A `short` takes up 16 bits, and thus can be one of exactly 65536 different values. ($2^{16} = 65536$.) The actual range is -32768 through 32767.
- Likewise, `int` (by far the most common) takes up 32 bits, and `long` is 64 bits. And the larger the type gets, the larger the range of values it represents
- Trade-off here! – If you want a larger range of values, you must sacrifice by using more memory. If you want to save memory, you must sacrifice by allowing yourself a smaller range of values.
- But in this course, we will basically stick with `int`.

The concept of a "literal" for a type

- We'll use the word *literal* to refer to the symbols you type directly into your program's code in order to indicate certain values.
- If you were to type `2 + 3` into your program's code, then the `2` and `3` you typed into your program are literals in this case.
- Integers you type directly into your program's code are literals of type `int`, rather than `short`, or `byte`, or `long`. That is one reason we tend to use `int` in this class rather than the other three integral types – it makes your life easier for now.
- Examples include `2, -56, 3451, -957`, and so on.

Floating-point types

- Two types: `float` and `double`
- These two types both refer to floating-point values, but of different size limits, such as the four different integral types had different size limits.
- A `float` uses up 32 bits of memory; a `double` uses up 64 bits of memory.
- Again, just as with integers, the larger the type gets, the larger the range of values it represents. Floating-point values, however, can have a larger range in two different ways:
  - They can have a larger *magnitude* (i.e. size of value).
  - They can have greater *precision* (i.e. number of significant digits).

  Values of type `double` can have both greater magnitude and greater precision than values of type `float`. So, we have the same same trade-off here as we did with integers! – a gain in value range results in using up more memory, and saving memory results in having a smaller value range available.
- Floating-point literals that you type into your program's code – symbols such as `98.6, 3.14159, -88420.22491,` and `0.000004556` – are literals of type `double`.
- In this course, we will basically stick with `double`, partly because the fact that literals are automatically of type `double` makes that type easier to deal with than `float` would be.
- One note of caution about floating point numbers – any numbers you get as a result of calculations are generally slighly imprecise, since computers need to round calculations just as you do. (For example, the square root of two is an irrational number and thus has an infinite number of digits; you need to round off somewhere.) As a result, often a floating-point result could be different than you expect in, say, the 12th significant digit. For that reason, you should NEVER compare two floating point numbers directly to see if they are equal; instead, you should be concerned with whether they are within, say, 0.00000000004 of each other (or within some other distance of each other). The syntax you learn in lecture packets 5 and 6 can help you with this.

`char`

- The values of type `char` are single characters.
- Literals of type `char` are single characters that appear within single quotes. For example: `'A'`, `'c'`, `'$'`, `')'`.
- A value of type `char` takes up 16 bits. That gives us the ability to store one of 65536 different characters. Why would need the ability to represent so many different characters, when your keyboard only has a hundred or so characters?
- The answer to that is: Java's character type is designed for international use – it can hold characters of many different languages, using a "international character to 16-bit bit string" translation standard known as Unicode.
- But, don't worry about Unicode right now. Just worry about putting single characters in single quotes as above.
- Warning!! When you are dealing with character literals in a program, don't forget the single quotes! You want to type `'c'`, not `c`, to indicate the literal character. If you leave off the single quotes it means something different (which we will explain shortly).

`boolean`

- There are only two values of the `boolean` type. The literals representing those values are `true` and `false`.
- Useful in situations where there are only two possibile results (ex: a certain part of the program either has completed its work or has not completed its work; a certain student either graduates or doesn't.)
- Needs exactly one bit of space – because we can simply say that the bit equalling 1 (i.e. the switch being on) inside the machine is equivalent to the boolean value `true`, and the bit being 0 (i.e. the switch being off) is equivalent to the boolean value `false`.

Summary of Type Information

type	size in bits	values
boolean	1	true or false
char	16	'\u0000' to '\uFFFF'
byte	8	-128 to 127 i.e. -(2^7) to (2^7) -1
short	16	-32768 to 32767 i.e. -(2^15) to (2^15) - 1
int	32	-2^31 to (2^31) - 1
long	64	-2^(63) to (2^63) - 1
float	32	-3.40292347E38 to 3.40292347E38
double	64	-1.79769313486231570E308 to 1.79769313486231570E308

Values in the machine

As we have previously discussed, all our data is really stored as bit strings in the computer's memory cells. In addition, we mentioned that a value that needs more than one cell will use consecutive cells. For example, a value of type `int` is 32 bits, which means it needs four 8-bit cells to hold all 32 bits. If the first 8 of those bits are stored at, say, address **a48**, then the next 8 bits are at address **a49**, the next 8 bits are at address **a50**, and the last 8 bits are in address **a51**.

Variables are abstractions of this idea – they allow us to think of our value as an item in a "labelled box", rather than thinking of it as a bit string spread across one or more consecutive memory cells. For that reason, thinking of how data is stored in the machine is not generally something you'll have to do – you can simply use the abstraction provided by variables, and let the environment (compiler, operating system, etc.) handle for you the particulars of where a variable's value should be stored in memory.

Variable Declaration Statements

- In Java, each variable can only hold one particular type of value, so before you can use a variable, you must announce which type the variable is supposed to hold.
- The statement we use to do this is known as a *variable declaration statement*.
- Format is: `typeName variableName;`
- Some examples:

```
int x;

char selectionLetter;

double temperature;

boolean isCompletedYet;
```

- Now `x` can only hold `int` values, `selectionLetter` can only hold `char` values, and so on.
- Note that we put single quotes around a characater literal – such as `'x'` – to distinguish it from a variable with that character as a name (such as `x` above).

Variable Assignment Statements

- Writing a value to a variable is known as *assignment*.
- Format of the statement is: `variableName = value;`
- Some examples, using the variables declared on the previous slide:

```
x = 2;
selectionLetter = 'c';
temperature = 98.6;
isCompletedYet = false;
```

- The very first assignment to a variable is known as *initialization* because it is the *initial* assignment.
- For example:

```
int myVal;
myVal = 7;    <--- this assigment is
                 an initializaton
myVal = 3;    <--- the 7 is replaced by 3,
                 this assignment is NOT an
                 initialization
```

- You can declare and initalize on the same line. For example:

```
int exam1 = 93;
```

Operators – (some of) the "verbs" of a computer language

Operators are symbols that indicate some kind of action is to be performed. These are usually very simple actions, such as:

- `+` : addition
- `-` : subtraction
- `*` : multiplication
- `/` : division
- `%` : modulus
- `++` : increment
- `--` : decrement
- `=` : assignment
- `()` : parenthesis

One symbol can mean different things in different contexts. This context depends on the type of the operands. For example, `a + b` is compiled to a different collection of machine language instructions depending on whether `a` and `b` are floating-point types, or integer types. In fact, `a` and `b` could be *different* types...and the language needs to have a rule about what happens in that case as well.

Operators have *precedence* – certain operators get evaluated before other operators. This is similar to algebra, where multiplication and division were performed left to right first, and once all the multiplications and divisions were completed, *then* addition and subtraction were performed, again from left to right. As in mathematics, the parenthesis have the highest precedence, so if you have an addition and a multiplication, and the addition is in parenthesis, then the addition comes first, even though the multiplication would be done first if the parenthesis were not there. For example, the expression `5 * (2 + 3)` equals `25`, even though the expression `5 * 2 + 3` equals `13` and the expression `3 + 2 * 5` equals `13`.

The `++` operator performs a very common operation – it adds `1` to the value of a variable. For example, if the variable `i` held the value `5`, then the expression `i++` would raise the value of `i` to `6`. Effectively, it performs an addition and an assignment together – adding one to the value inside `i`, and then storing that new sum back into `i`. Similarly, the `--` operator subtracts `1` from the value of a variable.

We will refer collectively to the addition, subtraction, multiplication, division, modulus, increment, and decrement operators (all the operators in the above list except the assignment operator and the parenthesis) as *arithmetic operators*, because they help us perform arithmetic. We will be exploring more operators in lecture packet 5, and any Java reference book probably has a full list of the operators and their precedence rules. Also, our course web page has a link to a website with that information.

Expressions – the "phrases" of a computer language.

An *expression* is a code segment that can be evaluated to produce a single value.

Expressions are not always complete units of work – in the cases where they are not, they are used in code segments that *are* complete units of work – and those code segments are generally trying to use the value that the expression evaluates to.

Expression examples:

- `6`
- `2 + 2`
- `exam1`
- `slope * x + intercept`
- `5280 * numberOfMiles`
- `total/numScores`
- `a + b - c + d - e`

The language itself defines many kinds of expressions in terms of other expressions. Meaning, if we make a list of what sorts of things constitute an expression, we could end up with a list much like the following:

- a literal
- a variable
- (some arithmetic expression) + (some other arithmetic expression)
- (some arithmetic expression) - (some other arithmetic expression)
- (some arithmetic expression) * (some other arithmetic expression)
- (some arithmetic expression) / (some other arithmetic expression)
- ...and so on...

In the list above, "arithmetic expression" refers to any expression that evaluates to a numerical type – i.e. to an `int` or `double` (or any of the other four numerical types that we won't use in CS125). For example, since `5` is a literal of type `int`, and so is `7`, both count as expressions, since as the above list indicates, literals are expressions (they evaluate to single values). However, that means `5 + 7` is also an expression, since you have two arithmetic expressions (`5` being the first one, and `7` being the second one) separated by a `+` operator. And since `3` is also a literal of type `int` and thus also an expression, then since `5 + 7` is an arithmetic expression, and `3` is an arithmetic expresssion, `5 + 7 - 3` is also an expression. And so on – you can use the fact that some expressions on the list above are defined in terms of other, smaller expressions, to create large expressions of aribtrary complexity. And each of them reduces to a single value – that's what makes them expressions!

A computer programming language (such as Java) doesn't need to have a rule for every single expression you might come up with. It only needs to provide the building blocks to let you create your own expressions, and primitives to start those rules off. In our list of possible expressions, the literals and variables are primitives, since those are very simple expressions that can't be broken down further. And then, a rule such as

```
(some arithmetic expression) + (some other arithmetic expression)
```

allows us to create a composition – we can create a larger expression out of smaller pieces. Then, that composition is itself an expression, and can be included in still-larger compositions. Every larger expression we create in this manner, is composed of smaller expressions (which are possibly primitive expressions – variables or literals) – and yet is itself still an expression, and thus is something that can be reduced to a single value.

Statements – the "sentences" of a computer language

A *statement* is a code segment that produces a complete unit of work. (I know that is a rather vague definition; as we learn more Java syntax, the concept will become clearer.)

We have already seen *variable declaration statements* and *assignment statements*. We said earlier that an assignment statement will have a value on its right-hand side; that value is actually an expression – and thus we could describe the syntax of an assignment statement as,
`variableName = expr;`, where `expr` is some expression that evaluates to a single value of the appropriate type. That value then gets stored in the variable on the left-hand side of the assignment statement.

Statement examples:

- `c = 2 + 2;`
- `x = 5;`
- `y = slope * x + intercept;`
- `int b;`
- `numberOfFeet = 5280 * numberOfMiles;`

We will be looking at *many* other kinds of statements besides declaration and assignment statements. Statements are the building blocks of a computer program - you list them one after the other, and they are run one by one until the program has completed.

So, we now know what types are, and we can declare variables of particular types and assign values to those variables, and we know how such statements should fit inside a Java program. So, as the final example for this packet, note the program below, in which we have taken some variable declarations and variable assignments and expressions involving the arithmetic operators, and placed them within a new program skeleton. Granted, the program below doesn't do too much that is interesting, but it will compile and it will run (producing no output whatsoever). In the next lecture packet, we will give you the abilty to input values from the user into your program, to use in your calculations, and we will give you the ability to print values (such as the results of your calculations) to the screen.

```
public class Example
{
   public static void main(String[] args)
   {
      int x, y, z;
      char selectionLetter;
      double temperature = 98.6;

      x = 2;
      selectionLetter = 'c';

      x = x + 3;
      y = x * 2;
      z = (x + y)/2;
      boolean isCompletedYet;
      isCompletedYet = false;

      x = 5;
      x = 0;
      int w = (2 * x) + (3 * y) + (y * z * 4);
      isCompletedYet = true;
      temperature = 44.5 + temperature;
   }
}
```