Variables and Expressions

We often think of programs as tools to transform data, starting with some input, doing calculations, and resulting in some output. So the first thing we need is a way to store our data.

A data structure is a tool in a program for storing data. The most basic version of this is a variable; later we will see more complex versions, for instance arrays for storing lists of data, or objects for storing collections of values and actions that represent a component of a situation.

Data Values

Remember that data is anything stored for a program other than the instructions: starting values, input from users or files, intermediate values, and results are all data. Some may be permanent characteristics or attributes in the world of the program, others may be a temporary current state that changes as the program runs.

All data is stored as binary patterns and the same binary pattern might sometimes be interpreted as a number, a text symbol, a color, a sound, a coordinate, et cetera.

In a program we sometimes work with *literal values*, where we just write a price, or name, or color into a program. A program that only uses literals is very limited; it only works for those specific values and if we want it to do something different we have to re-write the whole program -- if we used the literal 49.99 throughout our program whenever we needed to refer to the price of something, we have to update every mention of that in the program when the price goes up to 62.50.

At the hardware level, if each piece of data is stored on a row in Main Memory, we can use the Memory Address of the row to refer to the data instead of using the literal value. If the program is written this way, then we only have to change the value of that one location to change the way the whole program works.

This, however, requires thinking about hardware details and remembering a memory address, which is a number. Java provides us with named variables instead.

Variables

Variables allow us to associate a name with a memory location and use the name in the program. This will allow us to end up writing something more like

```
total = bonus + base_commission
```

instead of

```
    ADD MM103 MM104 MM105
```

which is certainly easier to read.

Variable Declaration and Assignment

Declaration is the term for a statement introducing a new variable into our program. When this happens, we are telling Java that we want to have space for a variable with that name, and it will take care of reserving that row (or rows) in MM and associating the name we choose with that MM address.

Remember that the same binary pattern is reused for numbers, letters, colors, etc.. To tell Java how to interpret the binary pattern, each variable in Java has a type, which also limits the possible values the variable can hold. Our first type will be int, which means the variable is used to hold an integer – a whole number. When declaring a variable, we must give the type as well as the name.

So, to tell Java that we want a variable called bonus that holds an integer, we would say

```
    int bonus;
```

Wait, is that code right? Don't we need a class and a main and… ? Yes we do! But when giving examples of statements in notes, I will leave those parts out for speed and readability. Until we start writing our own methods, you can assume that examples of code always live inside a main which lives inside a class (which is organized into a package).

Assignment is a statement giving a variable a value. We use a single equals sign to indicate assignment, and assignment will always be from right to left: the value on the right is being put into the variable on the left. So if we say

```
    int bonus; // declaration
    bonus = 70; // assignment
```

we mean that the value 70 is being stored in the variable called bonus. Under the hood, this means that Java looks up what row of main memory the bonus variable represents, and puts the binary pattern for 70 into that row in MM.

Notice that we didn't say int again, we only include the type in the original declaration, it would be a syntax error to repeat it later. Variables must be declared before they are assigned.

If we know the value of the bonus when we create the variable, we could do declaration and assignment all in one line:

```
int bonus = 70; // declaration and assignment
```

A variable lets us associate a name with a specific address in memory. Whenever we use that name in our code, it will (when the program is translated to run later) stand for the value in that memory location.

In java, variables that we create in methods like main do not have any value until we assign them one. Therefore it is a syntax error in Java to try to use these variables before assigning them. So

```
int bonus; // declared, but un-initialized
// syntax error, bonus has no value
System.out.println("Bonus is " + bonus);
// syntax error, bonus has no value
int total = 90 + bonus;
```

Variable Names

The text goes through all the rules for variable names in Java, but you have a lot of freedom for what names you use. It is very important to get in the habit of giving your variables names that will help you read and write your code.

Without being able to see the rest of the program it came from, do you know what this statement does? Is it correct to do the task it was written for?

```
a = b - ((c * b) + d + e);
```

Too hard to tell! This is code that won't let me be as lazy as I want to be!

Single letter variable names make sense if you are doing abstract math, or need a temporary variable to hold an intermediate value briefly while doing calculations.

But if you give all your variables short names thinking you are saving time by making them short to type, you probably aren't considering how much extra time you're going to spend trying to remember what b represents.

The problem isn't just the length, here's an even worse version:

```
var1 = var2 - ((var3 * var2) + var4 + var5);
```

Here is a more readable version. Now it is more clear what is going on, but if we're checking our math, we might still be unsure whether we're using the variables correctly

```
home = pay - ((tax * pay) + insure + retire);
```

And now a version that has longer variables, and makes it clear that the tax variable is a percentage, so it does need to be multiplied, while the insurance is a flat fee and does not.

```
takeHome = paycheck - ((taxPercent * paycheck) +
     insuranceFee + retirementContribution);
```

You will develop your own style for variable naming that is the best compromise between names that have enough information to make it clear what is going on and names that are short enough to easily read and type. Adding comments can help when variable names are bad, but it would always be better to fix the variable names so comments can explain more complicated issues.

Some new programmers start off too sloppy about variable names, using names like xyz and fklsjhu, which means their code is impossible to read once they've forgotten what those represented. Other new programmers freeze up every time they have to create a variable name and overthink it, which slows down their coding. For the moment, try for one-word names, and stick two together if you get stuck.

When we want to combine multiple words in a name in Java we usually put the parts next to each other but capitalize the later parts, like taxPercent or takeHome, called *camel casing*.

Java does not analyze your variable names for meaning. If you create a variable called taxPercent but you store the overall tax amount in it, the language will not give you an error.

Assignment

Assignment means putting a value into a variable. That value could be a literal value like 7, or a value currently stored in another variable, or the result of an operation like adding.

```
int cats = 17;
int dogs = cats; // since cats is 17, dogs is now also 17
int animals = cats + dogs; // animals is the result of
     adding, so 34
```

Assignment always goes from right to left. We refer to the Left Hand Side (LHS) and Right Hand Side (RHS) of an assignment. The LHS must be a variable, it doesn't make sense to say

```
6 = howManyPies;
```

because that would mean we are trying to change the value of 6!

The RHS of an assignment can be any expression that, when fully evaluated, comes down to a single value.

Setting equality through assignment makes a one-time copy of a value, it does not link the variables forever. If I set one variable equal to another, and then change the first one, that does not go back and change the value of the second:

```
Int cats = 17;
int dogs = cats; // copies value so dogs is 17
cats = 18; // cats has changed, but dogs is still 17
```

Variable Types

Remember that we re-use the same binary patterns for many purposes, since there are a limited number of possible combinations of ons and offs of a given length. So the same binary pattern might be used in one situation to store a whole number, in another situation as a fractional number, in another place as a letter of the alphabet, etc. In some cases we might need longer binary patterns to store special values, so we'd combine patterns on multiple rows of MM. So it would be helpful to have a way to tell the program how to interpret a pattern and how many rows in MM to use.

Type is how Java keeps track of this information for a variable. Each variable can only store one type of information and this tells how to interpret the binary pattern stored at that variable's location.

Java provides *primitive* types for variables that are built into the language and can be used in simple expressions: integers (whole numbers) doubles (real numbers, which can have a decimal part), characters (individual symbols for text, enclosed in single quotes) and booleans (true or false).

The huge power of object orientation is that it is easy to define our own types for variables, which we will spend most of the semester doing. These are called *object types*. One object typed that is built into the language and treated almost the same as the primitive is String. A String can hold sequences of characters for text. Most of the time it is much more useful to use a string than a char.

All of these are types we can write literal values for in our code.

keyword	Type	Description	example literal values
int	Integer	whole number	0, 1, 2, -3779
double	real	can have decimal place	0.001, -87.02

char	text symbol	letters, numbers, punctuation	‘a’, ‘Z’, ‘1’, ‘.’, ‘ ’[1]
boolean	boolean	true/false yes/no	true, false
String	Text	words, strings of text symbols	“a”, “ZZ”, “hi bye”, “ ”

Remember that to declare a variable, we put the type before the variable name, and then we don’t say the type again thereafter.

```
int numPuppies = 3;
double costOfPuppies = 59.99;
char puppy_letter = ‘p’;
boolean canPuppyFly = true;
String puppyName = “Mr. Fluffenstuff”;
```

Having typed variables tells Java how to interpret binary patterns, but just as importantly, it makes writing code easier by limiting possible values we can give to a variable. We said above that numPuppies is an integer; we want to have a whole number of puppies. Having 0.6 of a puppy is probably not good! If later I forget what I’m doing, and try to say

numPuppies = 0.6

Java will give me a syntax error.

Sometimes people seeing types for the first time feel like they are making programming harder. Now there are more errors you can get! But actually types are like guard-rails that take some of the burden of remembering variable types off of the programmer; you choose the type for a variable when you declare it, and after that the language will keep track of what values are allowed and not allowed. The error you might get isn’t a new burden you have to take on, it is a reminder that you already chose what kinds of values were appropriate for this variable.

If you now need a place to store some other type, the answer isn’t to try to stuff it into an existing variable. Just make a new variable of the type you need! There may be cases where you are programming in a situation where you have to limit the number of variables you declare, but for the most part, new programmers tend to be too stingy about declaring variables, so for right now, think of variables as cheap tools; you can make as many as you want.

[1] Technically, these curved quotes ‘x’ and “x” are different from straight quotes 'x' and "x" (which are actually foot and inch markings), they are stored in different binary patterns. If you type in a word processor like Word, it will often automatically curl your quotes to make the text more attractive and readable. But in general, programming languages only use straight quotes. If you copy an example from these notes, watch out for curly vs straight quotes!

Initialization

The first time a variable is assigned a value is called initialization. But what happens if we try to use a variable's value before it is initialized?

```
int cats = 17;
int dogs;
int animals = dogs + cats; //???
```

In Java, this is a syntax error. You must initialize a variable in a method like main before using it.

Constants

Sometimes we have a value in a program that will not change, so it is not "vary-able" but it is still a good idea to put it in a variable, since this means we don't have to remember the literal value and can use a name instead, and also this means that if this value ever is to change, we only have to change the line with the assignment, not everywhere the value appears in the program.

We call these *constants* and in Java we use the keyword `final` for them. We often give them all-caps names so they stand out.

```
final double BONUS = 8.99;
BONUS = 7.88; // error, cannot change
double baseComission = 14.50;
double total = BONUS + baseCommission;
```

Expressions

Instead of just giving each variable a literal value, we want to be able to have our programs do calculations for us. For algebraic operations, Java uses:

addition	+
subtraction	-
multiplication	*
Division	/

Remember that in mathematics multiplication and division have high precedence and addition and subtraction have low precedence. Within a level of precedence we work left to right, and we can adjust order of operations by adding parentheses.

We will also use + for *concatenation*, the operation that sticks strings together to make longer strings, often used when printing. This has the same precedence as for addition.

Any time I've said that some variable is assigned the sum of two other variables, I've been using operators in an expression.

An ***expression*** is a any combination of values using operators. So far we can write values as literals or get them from variables. An expression is evaluated and ends up as a single new value that we can then print or store in a new variable.

Java evaluates expressions by first replacing any variables by their values, and then working through all operators, in order of precedence (remember to go left to right at the same precedence level):

```
int cats = 17;
int dogs = cats + 1; // evaluation:
     //     17  + 1
     //     18

int animals = cats + dogs / 9 – 3 // evaluation:
     //        17  +  18 / 9 – 3 replace vars by values
     //        17  +    2    – 3 division high precedence
     //            19        – 3 left to right
     //                      16

String first = "Abe";
String last = "Aardvark"
System.out.println(first + " " + last); // evaluation
     //            "Abe" + " " + "Aardvark"   values
     //               "Abe "   + "Aardvark" concatentate
     //                 "Abe Aardvark"      concatentate
```

Casting

We should always choose the most appropriate types for the variables in our programs, but what happens when these types interact? In other words, adding two ints is fine, adding two doubles is fine, concatenating two Strings is fine… but what if I put the + operator between two variables or values of different types?

Well, for addition, we expect this to just work. We'd expect this total to be 12.3.

```
double total = 7.3 + 5;
```

And it is. This works in Java and does what we expect.

But if we think about what's happening under the hood, this is actually a really complex situation. 7.3 is stored as a double, using one set of rules for interpreting binary patterns, but 5 is stored as an int which uses a completely different set of rules!

The int 5 is *not the same* as the double 5.0 in Java even if they represent the same value in real mathematics. They might be stored in 64 bit hardware as.

5	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101
5.0	01000000 00010100 00000000 00000000 00000000 00000000 00000000 00000000

So, if Java tried to do math between an int and a double without taking this difference into account, we'd get incorrect results. However, this is the kind of hardware detail that programmers mostly don't want to think about, so Java is set up to automatically handle this problem for us!

When using algebraic operators on values of different types, Java changes one of them it so they are the same type first, and then does the math.

The process of changing a value from one type to another is called *casting*. Casting will at least always involve changing the binary pattern used to store the value into the system used for the variable's type. When Java does this automatically, it is ***implicit casting***.

```
double total = 7.3 + 5; // evaluation
     //        7.3 + 5.0 implicit cast to double
     //           12.3   addition of doubles
```

An int variable cannot hold the fractional part of a double, but any int has an equivalent double value. So any mathematical operation between an int and a double always casts the int to a double for any operation.

Note that this is also an issue on assignment, if I write

```
double d = 17;
```

I am triggering an implicit cast of the int 17 to the double 17.0.

But if we try to go in the other direction, we have a problem

```
int i = 7.3 + 5;
int ii = 17.0;
```

Again, an int cannot store the fractional part of a double, so both of these would be a syntax error, even though the double 17.0 doesn't *have* anything in its fractional part!

Java does provide a tool for deliberately changing the type of a value by ***explicit casting***. Put the type we want inside parentheses in front of the value whose type we want to change.

```
int num = (int)8.9 // becomes the integer value 8
```

Note that casting is not a mathematical operation, although in this case it does the same as taking the floor function of the number. Rounding is a mathematical operation that takes a double input and results in an integer output (in this case 8.9 would round to 9) , but that isn't what we are doing here, we are just throwing away the part of the data that won't fit into our representation system.

Integer Division

People are sometimes surprised by the need to cast to a different type to do arithmetic on numbers of different types. People are sometimes even more surprised that doing arithmetic on numbers of the same type does *not* automatically cast to a different type!

This is an error, because the result of the subtraction is still a double, 15.0 not 15

```
int i = 15.99 – 0.99;
```

This is not an error, because the result of the division is still an int!

```
int ii = 5 / 10;
```

Integer division always results in an integer. Since the integer cannot hold a fractional part, the value of ii after this statement is 0!

Forgetting how integer division works is a common reason for bad results. Suppose we have bought some fancy dog biscuits as treats for our puppies and we want to know how to cut them up so every puppy gets some:

```
int numPuppies = 7;
int numTreats = 2;
double treatsPerPuppy = numTreats/numPuppies;
```

Does this correctly give us the result? No, this says each puppy gets 0 because 2/7 is 0 in integer math. It does not matter that the variable on the LHS is a double because the expression on the RHS is doing math with ints and the implicit casting of integer to double happens too late.

```
double treatsPerPuppy = numTreats/numPuppies; // evaluation
     //                    2     /   7
     //                          0         integer division
     //                          0.0       implicit casting
```

New programmers usually think this means that they chose the wrong types for their variables and should have used doubles throughout, but that's not usually correct. Again, allowing fractional puppies is usually messy. And it may be that we are only able to purchase whole treats, so int may be the correct type to describe the situation for both of those values.

So how can we fix this, how can we tell Java we need to change the type of these values when doing the math? That's what explicit casting is for!

```
double treatsPerPuppy = (double)numTreats/numPuppies;
     //                  (double)  2     /   7
     //                  2.0     /   7   explicit cast
     //                  2.0     /   7.0   implicit cast
     //                  0.2857142857142857   double math
```

Notice we didn't have to cast both ints. Casting one explicitly triggered an implicit cast on the other. Make sure you understand why the above works but the following has the same problem as the original

```
double treatsPerPuppy = (double)(numTreats/numPuppies);
          // 0 treats per puppy!
```

Casting and Concatenation

We do have one more kind of operation we already know about, the other use of + to mean concatenation. It turns out we can use concatenation on types other than String, which we frequently do when printing.

```
int x = 5;
int y = 2;
String str = “abc”;
String str2 = str + str; // “abcabc”
String str3 = str + x // “abc5”
```

When we use the + operation between a String and an int, it cannot be addition, because that isn't defined for Strings, so it must be concatenation. so "abc" + 5 results in "abc5"

This is another case where the result is what we would expect, but the process of getting there is a lot more complex that people usually expect at first glance. Remember that 5 is stored as a binary pattern in memory. So is "abc"… except it is actually stored as multiple binary patterns, the ASCII for 'a' and 'b' and 'c' and also a lot of other structure related to the fact that String is actually a class, not a primitive type. So to concatenate these, Java first implicitly casts the int 5 to the String "5"

Even without concatenation, when we print numeric values, those values are implicitly cast to String, because humans don't usually like to read binary patterns.

```
System.out.println(x); // prints as 5, not binary
```

Suppose we now continue the code above and add the following statement. Try to picture what you think it will print.

```
System.out.println(x + y + str + y + x);
```

To get this right, you have to remember both String concatenation, and how Java evaluates expressions. Remember that concatenation + and addition + have the same precedence and within the same precedence, we go left to right. Let's work through it.

```
   x + y + str + y + x
// 5 + 2 + "abc" + 2 + 5
//   7   + "abc" + 2 + 5 int + int addition
//   "7"   + "abc" + 2 + 5 int + String triggers casting!
//       "7abc"    + 2 + 5 concatenation
//       "7abc"    + "2" + 5 casting
//              "7abc2"  + 5 concatenation
//              "7abc2"  + "5" casting
//              "7abc25"       concatenation
```

This is one reason that we usually don't do concatenation and math in the same statement. Beginners are sometimes tempted to do the last step of a calculation in the same statement as their final printout because they feel like they're making the code shorter and saving some work. But usually this puts you in a position for confusion like the above to give you a bad result.

Changing values

Variables often are changed over the course of a program. Most of the time, a variable's new value will be related to its old value, so we will often see statements where the same variable is on both sides of an assignment.

```
double sale = 8.00;
double coupon = 1.50;
sale = sale - coupon;
System.out.println("Final amount: $" + sale);
```

In the bold line, sale is on both sides of the equals sign. ***This does not mean you have to "solve" the equation!*** Remember that the RHS and LHS of an assignment are treated differently. We find the value of the RHS, and then change the variable on the LHS to hold that value. Also remember that when we evaluate an expression, we replace any variables by their values. So the bold line would be evaluated in the following steps

```
1. sale = sale - coupon
2. sale = 9.00 - 1.50 // replace variables by values
3. sale = 7.50 // do the operation
```

So now sale has a new value, which was based on the previous value. In speech, we would say that we "decreased sale by coupon" or just "subtracted coupon from sale." When we describe it this way, new programmers sometimes think they should write

```
double sale = 8.00;
double coupon = 1.50;
sale - coupon;
```

But that doesn't work! Remember that we should never be doing a calculation without storing the result. We need to put the result of that subtraction back into a variable, and in this case, it should go back into sale, replacing the previous value.

Make sure you understand why the following descriptions in comments would be written as the code below each (assuming all variables have already been declared and initialized):

```
//increase price by 5
price = price + 5;

// halve price
price = price / 2;

// triple price
```

```
    price = price * 3;

    //decrease price by rebate
    price = price - rebate;
```

Increasing by 1 is also called incrementing, and decreasing by 1 is called decrementing. These operations happen so often that we have special operators to write them more briefly

```
    //increment price
    price = price + 1;
    // increment operator
    price++;  // means same thing

    //decrement price
    price = price - 1;
    // decrement operator
    price--;  // means same thing
```