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. High level languages provide 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 introducing a new variable into our program. When this happens, we are telling the language 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. The language will handle knowing what the MM address is, so we don't have to.

Some languages do allow the programmer to access the actual MM address associated with a variable as a number and even do operations on that value while other languages hide this information. Having that access means we have more control over the specifics of what our program is doing at the hardware level, but it also means we have more power to do something that will mess things up[1]. When a language designer chooses what tools to include in a programming language, there is a tradeoff between power and ease of use. A language that provides more firepower also makes it easier to blow off your own foot.

One way in our pseudocode to declare a variable is just to put the keyword var in front of the name we want to use for the variable. So by saying

```
var bonus
```

we have told the language that from now on in our program, bonus will be the name of a variable.

Assignment is 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

```
bonus = 70
```

[1] Old-school programmers using languages with this kind of access sometimes did tricks like having one program leave data at a certain memory address and having another program simply read from that memory address, as a way of very efficiently moving data from one program to another, instead of having the first program save to a file on the hard drive and the second program read from that file. There are situations where resources such as space on a device or time to complete a task are very limited, where such approaches might really help. However, this relies on knowing exactly which MM addresses a program would be using every time, that no other programs would use or change those addresses. It also means that a small mistake might have one program overwrite the wrong data, or even an instruction in another program.

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

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

```
    var bonus = 70
```

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.

Variable Names

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.

Programming languages do not analyze your variable names for meaning. If you create a variable called tax_percent but you store the overall tax amount in it, the language will not give you an error.

Each programming language will have its own restrictions on how variables can be named. Variable names cannot be the same as a keyword that is part of the language, so we could not have a variable whose name is var, because we are already using var to indicate declaration, similarly we can't use something that's already a literal value, so we couldn't use 70 as the name of the variable. variables cannot have spaces in their names, so if we want to use multiple words in the name we might put a dash or underscore between the parts, or else put the parts next to each other but capitalize the later parts, called *camel casing*. some examples

```
    var number_of_puppies
    var cats-per-square-ironing-board
    var priceOfBeans
```

For our pseudocode, we will allow variable names to contain only: upper-case and lower-case letters, digits, underscores, and dashes, and nothing else. That way, if you see anything else, you know it cannot be part of a variable name. Most variables should use mostly lower-case letters and generally start with a lower-case.

Assignment

Assignment means putting a value into a variable, for which we use a single equal sign. 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.

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

In our pseudocode, as in most languages, 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:

```
cats = 17
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 languages keep 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.

The most common types are integers (whole numbers) doubles (real numbers, which can have a decimal part), characters (individual symbols for text, enclosed in single quotes) booleans (true or false), and strings (sequences of characters, enclosed in double quotes).

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

pseudocode keyword	Java keyword	Type	Description	example literal values
int	int	Integer	whole number	0, 1, 2, -3779
double	double	real	can have decimal place	0.001, -87.02
char	char	text symbol	letters, numbers, punctuation	‘a’, ‘Z’, ‘1’, ‘.’, ‘ ’[2]
boolean	boolean	boolean	true/false yes/no	true, false
string	String	Text	words, strings of text symbols	“a”, “ZZ”, “hi bye”, “ ”

Instead of using var, to declare a typed variable, we put the type before the variable name, and then we don’t say the type again thereafter.

```
int num_puppies = 3
double cost_of_puppies = 59.99
char puppy_letter = “p”
boolean can_puppy_fly = true
string puppy_name = “Mr. Fluffenstuff”

num_puppies = 7 // found more puppies, don’t say the type
     again
```

Having typed variables makes it easier for the language to know how to interpret binary patterns, but more importantly, it makes writing code easier by limiting possible values we can give to a variable. We said above that num_puppies is an integer; we want to have a whole number of puppies, having 0.6 of a puppy is probably not good! But if later I forget what I’m doing, and try to say

num_puppies = 0.6

I will get 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

[2] 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!

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.

To summarize, in our pseudocode: Declaring a variable in our pseudocode requires putting the type of the variable first, then the variable name. We will use a single equals sign for assignment, and assignment can be done on the same line as declaration. Once a variable is declared, we do not put the type in front of the variable again. If doing an example that requires variables not to be strongly typed, we will use var in place of the type, but this will not be used for most code.

Weakly Typed, Dynamically Typed, and Casting

The above explains how type works in most languages, which are *strongly typed*. Other languages, however, may take different approaches.

In a *dynamically typed* language, it *is* syntactically legal to assign a variable a value that does not match the variable's current type. When the program runs, the value will be changed into the correct type for the variable. 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. Weakly typed languages do a lot of *implicit casting* – the casting happens automatically without the programmer having to deliberately write code to make it happen.

For instance, in a dynamically typed language, we would be allowed to say

```
    string str = 10
```

But what value would str end up with when the program runs? The designers of the language would have to have determined how integer values are converted into strings. It could be that it simply becomes the string a human would use to type that integer value, "10". It could be that the binary to store that number would be reinterpreted as a character to be stored in the string; the number 10 can be stored as 00001010 in binary, and this happens to be the same as one common binary pattern for storing "A".

Strongly typed languages do usually include some implicit casting. If we are able to print numeric values, for instance, those values will probably be cast to string types, because humans don't usually like to read binary patterns. So if we say

```
    print 10
```

We would expect the number 10 to be implicitly cast to the string "10" so it could be printed in a way users understand, instead of seeing the binary pattern 00001010 for the value 10 printed.

The other most common implicit cast is from int to double. If we say

```
    double d = 17
```

It may not even register that we are using different types. For humans, 17 and 17.0 are obviously equivalent. But it is unlikely that under the hood, the binary pattern used to store 17 in the system for integers and the binary pattern used to store 17.0 in the system for real numbers would be the same binary pattern. Casting has to happen to convert from one to the other. Most strongly typed languages would do implicit casting here so that d1 ends up with the value 17.0.

A strongly typed language would not let us go in the other direction, however. If we try to put a real number into an int like this

```
    int num = 8.9
```

This would be a syntax error, because there is a decimal part to 8.9 that could not be preserved if we store it under the integer system. The strongly typed system put s up guardrails to avoid losing some of our data.

Strongly typed languages do often provide tools for deliberately changing the type of a value by *explicit casting*. In our psuedocode, we can explicitly cast by putting the type we want inside parentheses in front of the value whose type we want to change.

```
    int num = (int)8.9 // become 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 real 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.

Weakly typed languages could be said to respect the types of their variables and just do constant implicit casting. *Dynamically typed* languages change the type of a variable on the fly whenever it is assigned. So in a dynamically typed language, if we said

```
catval = “Fluffy”
print catval
catval = 3
print catval
catval = 8.2
print catval
```

Then the same variable would first be a string, then an int, then a double. Note that this is going to take more work for the language – when we print a value for catval, we have to go retrieve the binary pattern stored in the catval location in MM, and this value has to be interpreted appropriately, which means the language will have to keep track of whether it should evaluate the variable under the system for strings or for ints or for real numbers. Never having to think about the type of a variable can be very convenient and for some kinds of tasks it frees us up from thinking about all those implementation details like what format we are using for our binary patterns. But it does take off the guardrails that would stop us from using inappropriate values and means that it becomes our responsibility to check types ourselves.

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 // what value???
```

This will depend on the language you are working in. In some languages, this is simply a syntax error. For our pseudocode, this is what we will assume.

In some languages, all variable types have default values, with numbers generally defaulting to 0, Booleans to false, and other types to the special value `null`.

In some languages, however, the value of uninitialized variable is whatever binary pattern was sitting in the row in MM that the variable was allocated when it was declared. This “garbage value” could be a value from some other program that was using this part of memory earlier. It is almost never a good or useful value, so in these languages it is very important to remember to initialize your variables!

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 pseudocode we can use `const` or `constant` or, as in Java the keyword `final` for them, in addition to any type. We often give them all-caps names so they stand out.

```
const 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, we use:

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.

Later we will see other operators, for example relational operators such as > and < and logical operators like && and ||. In our pseudocode, operators generally are infix (remember this means they go between the operands).

Not all operators are defined on all types in all languages. In most languages it doesn't make sense to multiply a string or divide a boolean.

Expressions are evaluated 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;
```

This works in Java and does what we expect. The specifics may differ depending on language, but let's look at this in Java:

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 high level languages like Java are 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*. The Java syntax is to 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.

This is the kind of detail that in some cases is an important part of thinking through how our program will handle data, and in other cases is too fiddly a detail to consider when planning our approach to a large-scale problem. So, sometimes when writing pseudocode we would write this kind of detail out, in which case we'd borrow the Java syntax, while in other cases we might ignore types entirely. When writing a real program in Java, however, we have to get this right to end up with meaningful results.

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! We'll go a little further in the details of how not thinking through types can trip you up in a strongly typed language like Java:

The following 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, that is putting two strings next to each other to make a longer string. We use concatenation on many types other than string, for instance 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
print("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
```