Description

This week we'll see to create a Java project and program using NetBeans editor. We'll introduce the structure of a simple program including main() method. We will see the primitive Java data types. We will learn casting, constants and arithmetic operations.

Introduction

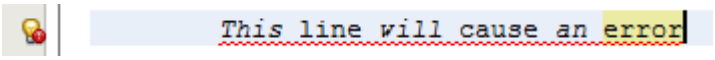
In this course, we'll be using NetBeans as our IDE (Integrated Development Environment). An IDE is a code editor that helps you while you type in your code with things like colour coding and code completion. It also provides an environment for you to compile, run, and debug your programs all in one place. NetBeans is an editor that is used industry wide, and is one of the most popular IDEs.

There are three steps in developing a Java program:

1. Write the program (called the source code) using an editor (such as NetBeans)
2. Compile it to produce an executable file (called *bytecode* in Java)
3. Run the program on a Java virtual machine (JVM)

The JVM translates the bytecode into something the underlying platform understands, and runs your program.

A nice feature of NetBeans is that you don't need to compile to see your errors. They will show with a red warning in the left margin of the source code window, and a red squiggly line under the part of the statement it doesn't like.



If you hover your mouse over the red dotted line a message will pop up that tells you what's wrong.

Text enclosed between /* and */ is called a *comment*. This is ignored by the compiler and has no effect on how the program runs. It's intended only for the benefit of human readers. Comments can be placed anywhere within a program.

A comment created using /* and */ can be one or several lines long. There is another way to create a single-line comment:

```
// A very simple Java program – the classic "Hello, World!"
```

Any text that comes after two forward slashes (//) is treated as a comment and is ignored by the compiler, up to the end of that line.

Why comments are important?

Comments don't affect how the program runs, and few programmers enjoy writing them, so it's tempting to leave them out. Try to resist this temptation. In this course, you will usually be writing relatively small programs, running them once, submitting some for marking, and then you're finished with them. Since they're simple programs and you've written them yourself and understand them completely, comments seem unnecessary.

As a working programmer, however, you'll be writing large, complex programs. These won't be thrown away after one use; they're likely to be in use for years. Other

programmers will likely need to modify the program you have written at some future date. Unless the program has been well commented, it will be a difficult task for them even to understand your program so they can work on it.

Java Classes

Java programs are made up of classes. In Java, the whole program (application) goes inside a class, which is saved into a source file with the extension .java.

One of the uses of a class in Java is to serve as a container for an application. This is not our primary definition of a class; however, we'll see other uses of classes soon. For now, just remember that a Java program must have at least one class definition.

A class definition begins with a *header*:

```
public class ClassName
```

"public" and "class" are Java keywords. That means they have special meaning to the compiler and can only be used in certain contexts. It's up to us to choose the class name.

Java is a case-sensitive language. That means that the compiler treats 'public" and "Public" as two different words.

The class header is followed by the *body* of the class or the *class definition*. The body always begins and ends with braces (curly brackets).

```
public class HelloWorld {
  // other lines go here
}
```

Notice how all of the lines that go inside the body of the class are **indented** relative to the header. Here again, the compiler does not care about this – it is done to make the program easier for humans to read. This does not seem very important for a small, simple program, but it will become increasingly important as we write larger programs with more complex structures, so get in the habit of always indenting.

Java Methods

There are several structures that can go inside the body of a class. For now, we're only going to work with one of them, a method. Think of a method as a group of one or more programming statements that collectively has a name. This is not the definition we'll actually use, but it's convenient for now. In our MathInJava program, we write the code for one method, named main. It's a special method because every complete Java application has to have a method named main. This is where the program begins its execution. For this program, our class has only one method: main.

Just as classes have a header and a body, so do methods. The header of the main method is

```
public static void main(String[] args)
```

As we've already seen, "public" is a keyword or reserved word that means something to Java. So are "static" and "void". One of the nice features of NetBeans is that keywords are purple. The following are all legal declarations for the main method:

static public void main(String[] args) OR static public void main(String bing[])

A method body begins and ends with braces. We position them in the same way as the braces of a class body.

Java Method Calls

"println" (short for "print line") is the name of another method. Unlike main, this is not a method we are writing ourselves. It is a method provided as part of the Java language - a predefined method. The Java language has thousands of them. When we write a line like:

```
System.out.println("Hello, World!");
```

we are *calling* (or *invoking*) the method. It can also be called "*sending a message to*" System.out. In Java, System.out is the console, which is an easy place to display the output from our program. (Note that System.out is an unusual name, we won't normally have a dot in the middle of the name of a class or object.)

However, we always put a dot after the class or object name and before the method name. When the program executes, calling a method makes the computer carry out whatever tasks the method performs. println( ) prints text and moves to a new line after the text is printed. "System.out" tells the computer where to print the message (to the console), and the text inside the parentheses and the double quotation marks, "Hello, World!" tells it what to print.

The complete line: `System.out.println("Hello, World!");`

is a *statement*. You'll notice that the statement ends with a semicolon. Our main method body has only one statement, but method bodies can have many statements.

The code completion feature gives you a lot of information about the various methods. The header of the println method definition looks like this:

```
public void println(String x)
```

void is the return type of the method. **When the return type of a method is void, the call to the method is coded as a separate statement.**

The thing in the brackets is a **parameter** (String x). We can use **parameters** to supply data to the method. Parameters make the method more flexible and useful in different situations. To call this method, we need to pass a String type variable to it. There are other versions of this method that take different data types (such as a boolean or a char) as a parameter.

When we want to execute a method, we "call" it. Instead of the term calling, you may hear any of these: invoking, executing, or sending a message to. When we write a method call such as

```
System.out.println("Hello, World!");
```

the code in the brackets is called an **argument** ("Hello, World!"). The argument list does **not** include the data types of the arguments (in this case String).

The value of the **argument** in the method call ("Hello, World!") is assigned to the **parameter** in the method definition when control is passed to the method, so the x in String x becomes "Hello World" when the method gets executed.

Note that, in the method **call**, the items in the brackets are actually called **arguments**. In the **definition** of the method itself, they are called **parameters**. The number and

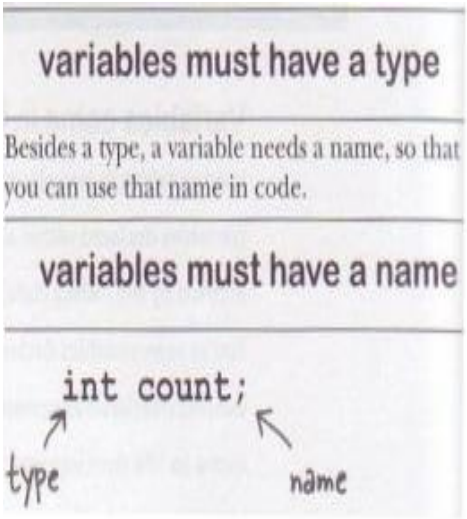
data types of the **arguments** in the method call must match the number and data types of the **parameters** in the method definition. Since the println method definition has one parameter of type String, a call to this method must have one argument of type String.

Variables

A variable is a location in memory that can be used to store data. Before you can use a variable, you need to *declare* it. The lines

```
double radius;
int numGears;
```

declare two variables. A variable declaration consists of a data type and a name.



The data type (double, and int) tells the computer what sort of data can be stored in the variable, and how much memory should be used for this variable.

Java has hundreds of predefined types that are useful in programming. These types are divided into two categories:

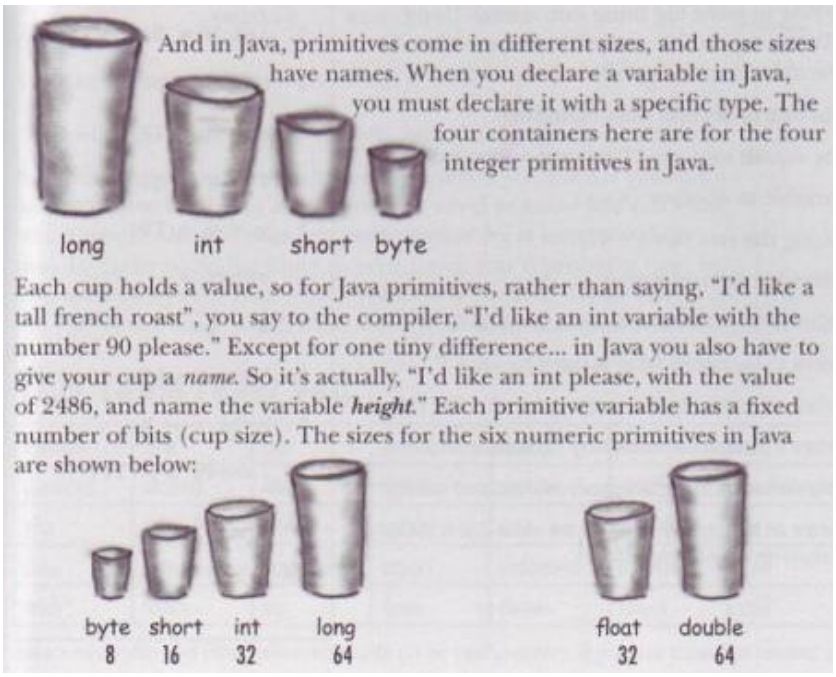
- The *primitive types* store various kinds of numbers and can be used as building blocks to build other types. There are eight of them. They all start with a lower-case letter. We cannot create new primitive types. More about these in a moment.
- The *reference types* store references to *objects* or instances of classes. Reference types are sometimes called *class types* (as in the Gaddis text) or *object reference variables* because they are the names of classes. System and Math are examples of reference types. They all start with a capital letter. We can create new reference types by building classes of our own, which we will do later. We will come back to reference variables next week.

Names (Identifiers)

The rules for choosing a name are given here http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html under Naming. This also explains the conventions you should follow. To summarize, variable names are case-sensitive, and should start with a letter. Spaces are not allowed, and underscores shouldn't be used except in special cases.Variable names should be short yet meaningful.

If you break these rules, the compiler will give you an error message:

```
11      int oneNumber;
        int *number;
        int 1number;
        int one number;
```



And in Java, primitives come in different sizes, and those sizes have names. When you declare a variable in Java, you must declare it with a specific type. The four containers here are for the four integer primitives in Java.

Each cup holds a value, so for Java primitives, rather than saying, "I'd like a tall french roast", you say to the compiler, "I'd like an int variable with the number 90 please." Except for one tiny difference... in Java you also have to give your cup a *name*. So it's actually, "I'd like an int please, with the value of 2486, and name the variable ***height***." Each primitive variable has a fixed number of bits (cup size). The sizes for the six numeric primitives in Java are shown below:

Primitive Types

Java has eight built in data types, called primitive types. ("Primitive" means these are the most basic data types, which are starting points for building other types.)

Primitive Types

Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32 bits	varies
double	64 bits	varies

Six types are used to store numbers. We can divide these into two groups. Four types, **byte, short, int** and **long**, are used to store integers (i.e. whole numbers with no fractional part, like 0, 1, 2, 3, 33520564, -7453, but not 1.5 or -99.99999). The difference between these types is how much memory each one uses, and what is the largest value that can be stored in each. You don't need to learn the exact maximum value for each data type, but it's a good idea to have a general idea of them.

The other two types, **float** and **double**, are used to store numbers with decimals, or floating-point numbers, i.e. ones that may have a fractional part. double requires more memory, but can store larger numbers than float. More importantly, double stores numbers with greater precision.

As a rule of thumb, use int for integers and double for floating-point numbers. Use long for very large integers. Don't bother with byte, short and float (you may occasionally need them).

"integer types" have exact precision, but floating point types do not. You know that if you compute 1/3 as a decimal fraction, it repeats forever 0.333333333….. We can't store an infinite number of digits in the computer's memory, so we have to approximate by only keeping a finite number of digits. double keeps more than float. We usually use double for floating-point types, and only use float in special cases.

By the way, in these notes, be aware that "integer types" means any of byte, short, int and long, not just int. "Floating-point types" means either of float or double, not only float.

Note also that these data type names are Java keywords and are all lower case. We'll see later that there are Java classes with similar names, but starting with upper case, like Byte, Integer, or Double. We don't want to use these yet. **Write your data types in lower case.**

Data type char

A seventh primitive type, named *char*, is used to store individual characters, such as letters, punctuation, etc. Here are some statements that declare char variables and assign values to them.

```
char letter;
letter = 'A';
char questionMark = '?';
char digit = '7';
```

char literals must be enclosed in single quotation marks (not double quotes). If we write A without the quotation marks, the compiler will assume this is a variable name, and if no such variable has been declared, it will generate an error message.

As shown above, digits 0, 1, 2, 3,…, 9 can be stored in a char variable. The character 7 is not the same as a number 7 of type int, double, etc. A char '7' can be printed, but you would not use it in arithmetic calculations in the way that you would use an int value of 7.

We'll look at booleans later.

Assignment Operator

The single equals sign (=) is called the *assignment operator* in Java. Keep in mind that this works differently from equals in mathematics. In math, we can write either:

1 + 1 = 2

or:

2 = 1 + 1

These tell us exactly the same thing. This is not the case in Java.

The Java assignment operator tells the computer to take a value that is on the right hand side of the assignment operator, and put it in the variable (i.e. in a certain location in memory) on the left hand side of the operator. The things on either side of the operator are called *operands*. The assignment operator requires two operands, so it is called a *binary operator*.

The left-hand operator must always be a variable name. The right-hand operator could be a number, a variable, or an arithmetic statement. For example:

```
int x;
int y;
int z;
x = 3; // assigns the value 3 to the variable x
y = x; // assigns the value already in x (3) to the variable y
z = x + y; // Computes x + y (3 + 3) and assigns the
           //result (6) to the variable z
```

Unlike in math, we cannot write:

```
int x;
int y;
int z;
3 = x; // left-hand operand must be a variable
x + y = z; // left-hand operand must be a single variable,
           // not an arithmetic expression
```

Notice how, in all these examples, I always put a space on either side of the assignment operator:

x = 3;
not
x=3;

This is not required by the rules of the language. Either of the above lines would compile correctly and would give the same result. The spaces are to improve readability for humans. It's good style to put them in, so get in the habit of always doing so.

You are allowed to do assignments inside larger expressions, such as:

System.out.println(x = 1);

It is perfectly all right to combine a variable declaration and assignment of an initial value into a single statement. You can write either:

int x;
x = 1;

or

int x = 1;

Math

As we have already seen in some of the examples, Java has operators that allow you to do arithmetic. The five arithmetic operators are +.-. *, / and %. Addition, subtraction and multiplication work just like in elementary school arithmetic, or on your pocket calculator. Note that the symbol for multiplication is the asterisk:*, not an 'x' as used in school arithmetic.

The symbol for division is the forward slash: / . Don't confuse this with the backslash: \. In Java, if you are working with numbers with decimals (floating-point numbers, which in Java would be called floats or doubles), division works the same way as in arithmetic or your pocket calculator. For example, after executing the following lines:

```
double x = 29.0;
double y = 10.0;
double z = x / y;
```

z will have the value 2.9.

Integer Division

Division using integers, however, works differently. If we divide one integer by another, the result is also an integer. For example, if we execute

```
int i = 29;
int j = 10;
int k = i / j;
```

then k will have the value 2, not 2.9. The fractional part of the result is discarded. Note that the result is **not** rounded off to the nearest integer (3, in the case above).

Remainder Operator (Modulo Operator)

Java has a fifth arithmetic operator that you have probably not seen before, called the remainder or modulo operator. Its symbol is the percent sign: %, but **it has nothing to do with percentages**. The remainder operator is related to integer division. As its name says, it represents the remainder after we perform a division. In the above example, 10 "goes into" 29 two times, so 29 / 10 gives 2, but that's not the end of the story. There is still 9 "left over" after the division is performed, so 29 % 10 gives 9.

Java does not forbid you from using the remainder operator with floating point numbers, but this is hardly ever useful. In practice, we only use it with int variables.

Operator Precedence

Here's a traditional operator precedence chart to remind you. The operations at the top of the chart are done before the ones further down:

Operator Precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Shorthand Operators

In computer programming, it is very common to take a value from a variable, perform an operation on it, and assign the result to that same variable. For example,

```
int i = 2;
i = i + 5;
```

In the last line above, we must evaluate the expression on the right hand side of the assignment operator, then put this value into the variable on the left hand side. On the right hand side, i has the value 2 and 2 + 5 gives 7, so we put 7 in i, changing the value in this variable.

Because it's so common to use the same variable on both sides of an assignment operation, Java provides you with a shorthand way of writing this. Instead of

```
i = i + 5;
```

we can write

```
i += 5;
```

These two expressions mean exactly the same thing. Note that there is no space between the + and the = in the shorthand operator.

There are shorthand assignment operators for all of the arithmetic operations: +=, -=, *=, /= and %=. The ones for addition and subtraction are much more commonly used than the ones for multiplication, division and remainder, but there's nothing wrong with using the latter ones.

Increment and decrement Operators

Very often, we want to increase the value in a variable by one. We could write:

```
a = a + 1;
```

or

```
a += 1;
```

but Java provides us with two even shorter ways of performing the same task:

```
++a;  or  a++;
```

The two plus signs together are a unary operator, called the increment operator. They increase the value in the variable they operate on by 1. (By convention, we do not put a space between a unary operator and its operand.)

The increment operator can be written either before or after its operand. If this operator is used by itself in a complete statement, as in the two examples above, it works exactly the same whether written before or after the operand. However, it works differently when this operator is used in a larger expression:

```
// fragment A (postincrement operator)
int i = 10;
int newNum = 10 * i++;
```

is equivalent to:

```
// fragment B
int i = 10;
int newNum = 10 * i;
i = i + 1;
```

so i has the value 11 and newNum has the value 100 after either of the above code fragments are executed, while:

```
// fragment C (preincrement operator)
int i = 10;
int newNum = 10 * ++i;
```

is equivalent to:

```
// fragment D
int i = 10;
i = i + 1;
int newNum = 10 * i;
```

so i has the value 11 and newNum has the value 110 after either fragment C or D is executed.

In summary, if the increment operator comes **before** its operand, you do the increment **before** you evaluate the larger expression. If the increment operator comes **after** its operand, you do the increment **after** you evaluate the larger expression.

Many programmers like to take advantage of this operator in larger expressions to write more compact expressions.

The decrement operators, --a and a--, work exactly like the increment operators, except that they decrease the value in a variable by one.

The Java API Math class

Math is a class in the standard Java library that contains methods for performing mathematical operations. All of these methods are static, which means you don't need an object to call them. Instead, you call them by preceding the method name with the class name, Math. You don't need to declare or import anything to use this class.

You are not expected to memorize the details of all the methods in Math. When you need to know about a method, you can use the code completion in NetBeans to get the details of a method.

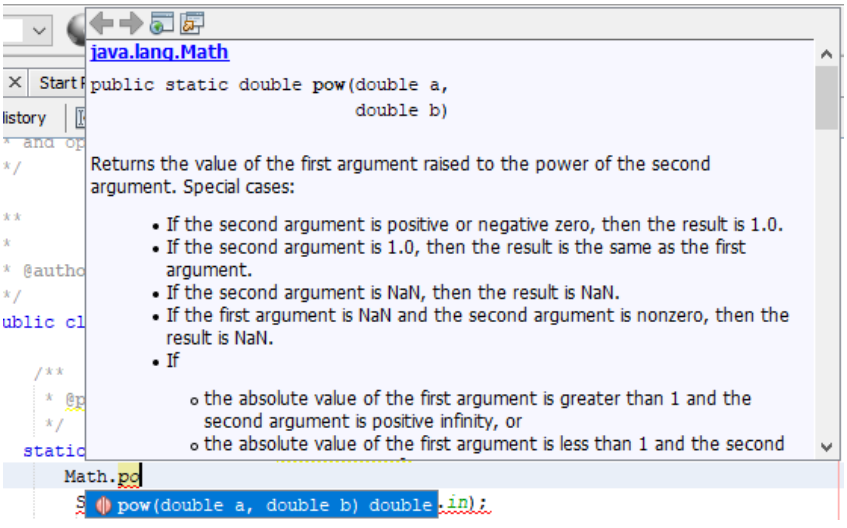
Or you can look it up in the on-line documentation called the Java API (Application Programming Interface Specifications).

Go to https://docs.oracle.com/en/java/javase/17/docs/api/index.html

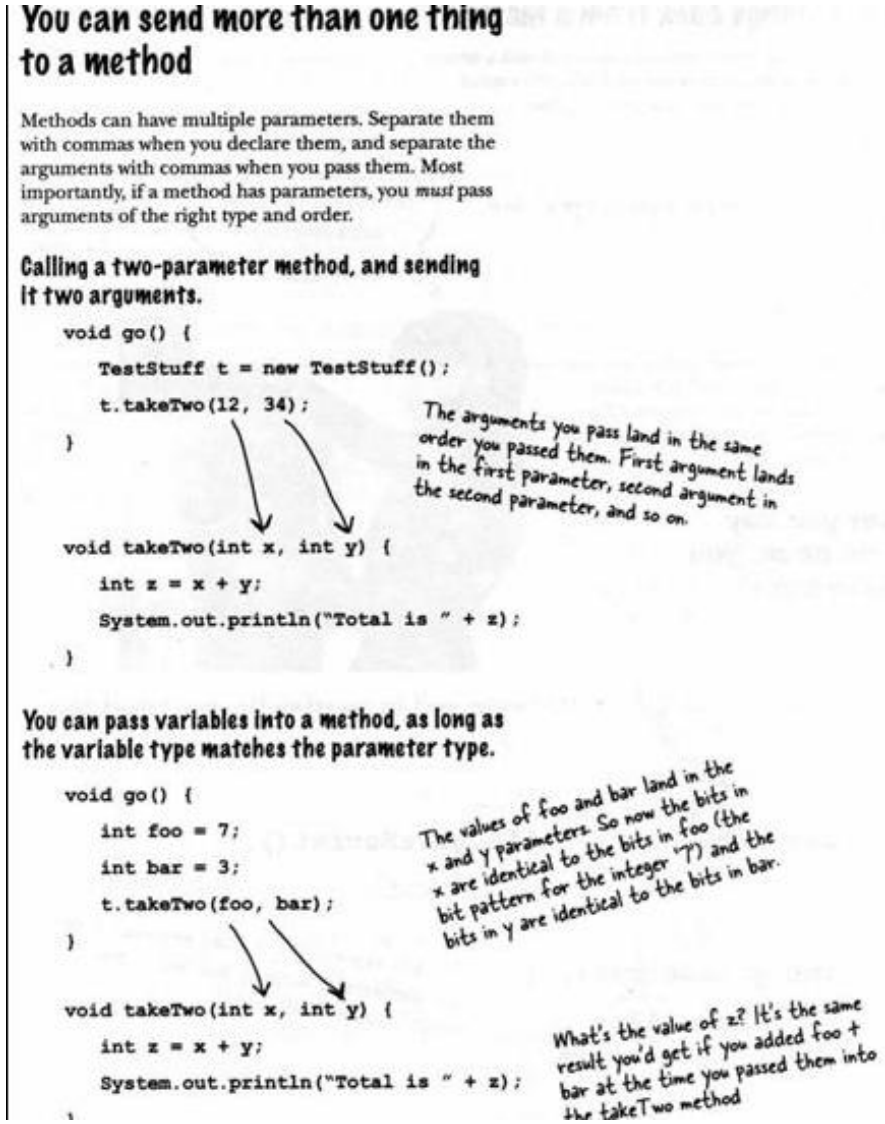
Type "Math" in the search bar and click on this name. A list of all of the methods in this class will appear in the right-hand frame.

Multiple Parameters

A simple example of a Math method is one called "pow", to raise a number to a power. Java does not have an operator to do this; it uses a method to perform this task. Here's the code completion for the pow method. You can see that the method is static (it doesn't need an object to call it, just the class name). You can also see that it takes two parameters, both doubles. Remember that you need to specify the arguments in the method call - the arguments and parameters have to be the same in how many there are, their data types, and the order they are specified:



Here's another interpretation of passing more than one parameter:



You can send more than one thing to a method

Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them. Most importantly, if a method has parameters, you *must* pass arguments of the right type and order.

Calling a two-parameter method, and sending it two arguments.

```
void go() {
   TestStuff t = new TestStuff();
   t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

You can pass variables into a method, as long as the variable type matches the parameter type.

```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer "7") and the bits in y are identical to the bits in bar.

What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method

Non-Void Methods

But, back to the Math class. You can also see from the Math.pow image above that the return type of this method is not void, but double; this is a non-void method.

After the method executes and returns this value, we usually want to do something with it.

The most common thing to do with a return value is to assign it to a variable. Here's a code snippet that demonstrates this, as well as demonstrating two parameters:

```
double a = 7.0;
double b = 3.0;
double c = Math.pow(a, b);
```

The return value of the Math.pow method is assigned to c.

Another common thing to do with a return value is to use it as an argument to another method call. (In the statement below, you'll notice + signs between the things we want to print. In this case, the + don't mean addition, they mean "concatenation", which means joining two strings together.) For example:

```
System.out.println(a + " to the power " + b + " = " + Math.pow(a, b));
```

By the way, if you want to raise a number to a small integer power (such as 2, to square a number or 3, to cube it) it is much more efficient to use multiplication, rather than the pow method. If you have a variable x and you want to square it, do

```
double x = 2.3;
double y = x * x;
```

instead of

```
double x = 2.3;
double y = Math.pow(x, 2);
```

Both give the same result, but the first is both easier to code and faster to run.

Another "by the way": excessive abbreviation of method names is not considered a good practice. If the "pow" method were being developed today, it would probably be named "power". Similarly, "sqrt" would probably be named "squareRoot". Names like "pow" and "sqrt" were used when the C language was developed in the 1970s, when extreme abbreviation was in fashion. Java has borrowed these names unchanged from C.

You can generate random numbers with the method Math.random(), but Java provides us with an easier way to perform these tasks, which we'll study in a later lesson.

Type Casting

We saw previously that there are six different primitive types for storing numerical data, and the largest value that can be stored in a variable is different for each of these types. If we sort these types by their maximum value, then we have, from smallest to largest:

```
byte, short, int, long, float, double
```

If we have an arithmetic operation with operands of two different types, the operands will be converted so that both are of the “larger” type, according to the ordering above, and the result of the operation will also be of this type. So, if we have:

```
int a = 3;
double b = 1.5;
double c = b / a;
```

b / a involves two different data types, so both are converted to the larger type, in this case double, before we do the division. The result of the division is also a double.

A value of a “smaller” data type (based on the list above) can be assigned to a variable of a “larger” type.

There is no problem with writing:

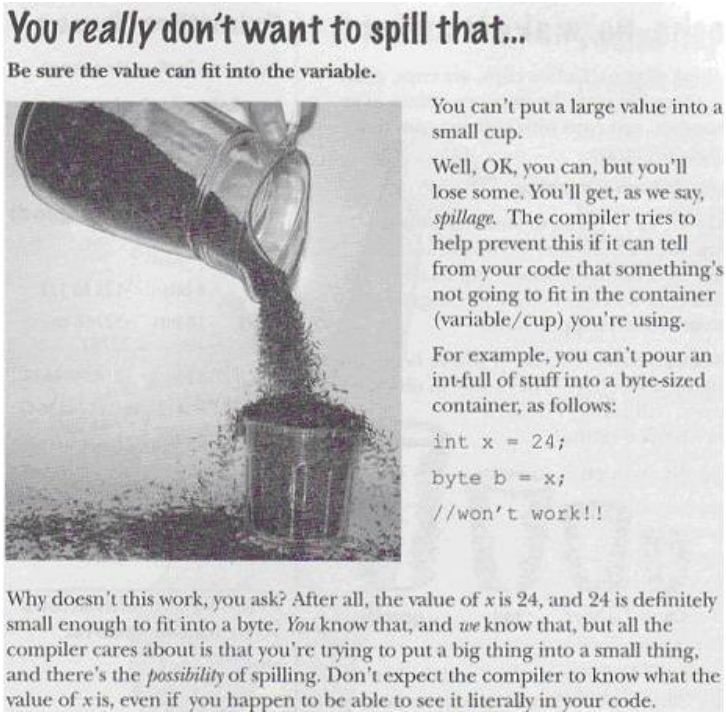
```
int a = 3;
double b = a;
```

This assigns a value of 3.0 to b.

However, if you try to assign a value of a “larger” data type to a variable of a “smaller” type, the program will not compile. If you write

```
double b = 3.6;
int a = b;
```

the compiler will give you an error message, saying “Possible loss of precision”. The problem here is that some of the information in the double variable might be lost when you try to put it into an int variable. In the case above, you can’t put the fractional part of 3.6 in an int. In other cases, the value in the double might be too large to store in an int (for example 3E10). The compiler is trying to protect you against making an error by not allowing assignments from a larger to a smaller type.



You *really* don’t want to spill that...

Be sure the value can fit into the variable.

You can’t put a large value into a small cup.

Well, OK, you can, but you’ll lose some. You’ll get, as we say, *spillage*. The compiler tries to help prevent this if it can tell from your code that something’s not going to fit in the container (variable/cup) you’re using.

For example, you can’t pour an int-full of stuff into a byte-sized container, as follows:

```
int x = 24;
byte b = x;
//won't work!!
```

Why doesn’t this work, you ask? After all, the value of *x* is 24, and 24 is definitely small enough to fit into a byte. *You* know that, and *we* know that, but all the compiler cares about is that you’re trying to put a big thing into a small thing, and there’s the *possibility* of spilling. Don’t expect the compiler to know what the value of *x* is, even if you happen to be able to see it literally in your code.

If you don’t care about this risk, you can overrule the compiler and tell it to go ahead and convert the larger data type to a smaller type. This is called **casting**. You do it by writing the new data type in parentheses before the value you wish to convert. For example:

```
double b = 3.6;
int a = (int)b;
```

This takes the value in b, and converts it to an int before assigning it to a. (The value in b itself is not changed. It remains a double.) An int cannot hold a fractional part, so this part is discarded, and 3 is assigned to a. Note that casting a floating-point number to an integer type does **not** round it off to the nearest integer – it just truncates the fractional part. In the example above, a gets the value 3, not 4.

Constants

A constant is a value that won't change during your program's execution, things like the minimum a person should pay towards their credit card balance. A constant is like a variable in that it needs a name and a datatype, and it takes up memory. You can turn a variable into a constant by putting the keyword “final” in the declaration. Once you have assigned a value to a constant, you cannot change it later in the program.

By convention, the names of constants are written in all upper case letters. If the name has more than one word, the WORDS_ARE_JOINED_BY_UNDERSCORES.

It is a very good idea to use constants with meaningful names, rather than scattering numbers through the body of your program. For example, if you are writing a program to compute sales tax, you might have a line of code like:

```
federalTax = 0.06 * price;
```

In a complex program, you might have several places where this same value 0.06 appears. It is much better to write:

```
final double FEDERAL_TAX_RATE = 0.06;
  // ... other lines ...
federalTax = FEDERAL_TAX_RATE * price;
```

This makes it much clearer to the reader why you're multiplying by 0.06. What's more, if the tax rate changes, you only have to change one line in your program. You don't have to search through the entire program, looking for all the places where 0.06 occurs.

Literals

A number that appears directly in a program is called a literal. If a literal has a decimal point (e.g. 5.4), then the literal is treated as a double. If you want to indicate that this number should be treated as a float, then you add the letter F: 5.4F (either upper or lower case).

```
double x = 5.4;
float y = 5.4F;
```

Numbers in scientific notation are also treated as doubles. Numbers in scientific notation can be written with or without a decimal point:

```
double x = 3E6;
```

This means 3×10^6.

If a literal has neither a decimal point nor an exponent, then it is treated as an integer. By default, it is treated as an int. If you want it to be treated as a long, you append the letter L (either upper or lower case).

```
Long i = 1234567890123L;
```