1.2 Built-in Types of Data

When programming in Java, you must always be aware of the type of data that your program is processing. The programs in Section 1.1 process strings of characters, many of the programs in this section process numbers, and we consider numerous other types later in the book. Understanding the distinctions among them is so important that we formally define the idea: a *data type* is a *set of values* and a *set of operations* defined on those values. You are familiar with various types of numbers, such as integers and real numbers, and with operations defined on them, such as addition and multiplication. In mathematics, we are accustomed to thinking of sets of numbers as being infinite; in computer programs we have to work with a finite number of possibilities. Each operation that we perform is well-defined *only* for the finite set of values in an associated data type.

Programs in this section

There are eight *primitive* types of data in Java, mostly for different kinds of numbers. Of the eight primitive types, we most often use these: `int` for integers; `double` for real numbers; and `boolean` for true-false values. There are other types of data available in Java libraries: for example, the programs in Section 1.1 use the type `String` for strings of characters. Java treats the `String` type differently from other types because its usage for input and output is essential. Accordingly, it shares some characteristics of the primitive types: for example, some of its operations are built in to the Java language. For clarity, we refer to primitive types and `String` collectively as *built-in* types. For the time being, we concentrate on programs that are based on computing with built-in types. Later, you will learn about Java library data types and building your own data types. Indeed, programming in Java is often centered on building data types, as you shall see in Chapter 3.

After defining basic terms, we consider several sample programs and code fragments that illustrate the use of different types of data. These code fragments do not do much real computing, but you will soon see similar code in longer programs. Understanding data types (values and operations on them) is an essential step in beginning to program. It sets the stage for us to begin working with more intricate programs in the next section. Every program that you write will use code like the tiny fragments shown in this section.

type	*set of values*	*common operators*	*sample literal values*
`int`	integers	`+ - * / %`	`99 -12 2147483647`
`double`	floating-point numbers	`+ - * /`	`3.14 -2.5 6.022e23`
`boolean`	boolean values	`&& || !`	`true false`
`char`	characters		`'A' '1' '%' '\n'`
`String`	sequences of characters	`+`	`"AB" Hello" "2.5"`

Basic built-in data types

Definitions To talk about data types, we need to introduce some terminology. To do so, we start with the following code fragment:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

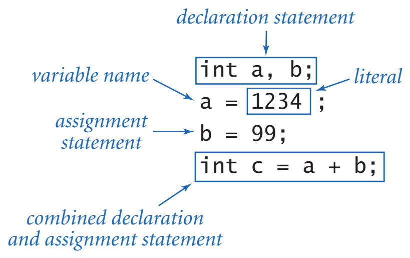
The first line is a *declaration* that declares the names of three *variables* to be the *identifiers* `a`, `b`, and `c` and their type to be `int`. The next three lines are *assignment statements* that change the values of the variables, using the *literals* `1234` and `99`, and the *expression* `a + b`, with the end result that `c` has the value `1333`.

Identifiers. We use identifiers to name variables (and many other things) in Java. An identifier is a sequence of letters, digits, `_`, and `$`, the first of which is not a digit. The sequences of characters `abc`, `Ab$`, `abc123`, and `a_b` are all legal Java identifiers, but `Ab*`, `1abc`, and `a+b` are not. Identifiers are case-sensitive, so `Ab`, `ab`, and `AB` are all different names. You cannot use certain *reserved words*—such as `public`, `static`, `int`, `double`, and so forth—to name variables.

Literals. A literal is a source-code representation of a data-type value. We use strings of digits like `1234` or `99` to define `int` literal values, and add a decimal point as in `3.14159` or `2.71828` to define `double` literal values. To specify a `boolean` value, we use the keywords `true` or `false`, and to specify a `String`, we use a sequence of characters enclosed in quotes, such as `"Hello, World"`. We will consider other kinds of literals as we consider each data type in more detail.

Variables. A variable is a name that we use to refer to a data-type value. We use variables to keep track of changing values as a computation unfolds. For example, we use the variable `n` in many programs to count things. We create a variable in a *declaration* that specifies its type and gives it a name. We compute with it by using the name in an *expression* that uses operations defined for its type. Each variable always stores one of the permissible data-type values.

Declaration statements. A declaration statement associates a variable name with a type at compile time. Java requires us to use declarations to specify the names and types of variables. By doing so, we are being explicit about any computation that we are specifying. Java is said to be a *strongly-typed* language, because the Java compiler can check for consistency at compile time (for example, it does not permit us to add a `String` to a `double`). This situation is precisely analogous to making sure that quantities have the proper units in a scientific application (for example, it does not make sense to add a quantity measured in inches to another measured in pounds). Declarations can appear anywhere before a variable is first used—most often, we put them *at* the point of first use.



```
int a, b;
a = 1234 ;
b = 99;
int c = a + b;
```

Using a primitive data type

Assignment statements. An assignment statement associates a data-type value with a variable. When we write `c = a + b` in Java, we are not expressing mathematical equality, but are instead expressing an action: set the value of the variable `c` to be the value of `a` plus the value of `b`. It is true that `c` is mathematically equal to `a + b` immediately after the assignment statement has been executed, but the point of the statement is to change the value of `c` (if necessary). The left-hand side of an assignment statement must be a single variable; the right-hand side can be an arbitrary *expression* that produces values of the type. For example, we can say `discriminant = b*b - 4*a*c` in Java, but we cannot say `a + b = b + a` or `1 = a`. In short, *the meaning of = is decidedly not the same as in mathematical equations.* For example, `a = b` is certainly not the same as `b = a`, and while the value of `c` is the value of `a` plus the value of `b` after `c = a + b` has been executed, that may cease to be the case if subsequent statements change the values of any of the variables.

Initialization. In a simple declaration, the initial value of the variable is undefined. For economy, we can combine a declaration with an assignment statement to provide an initial value for the variable.

Tracing changes in variable values. As a final check on your understanding of the purpose of assignment statements, convince yourself that the following code *exchanges* the values of `a` and `b` (assume that `a` and `b` are `int` variables):

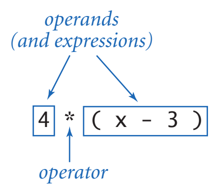
```
    int t = a;
    a = b;
    b = t;
```

To do so, use a time-honored method of examining program behavior: study a table of the variable values after each statement (such a table is known as a *trace*).

	a	b	t
`int a, b;`	*undefined*	*undefined*	
`a = 1234;`	1234	*undefined*	
`b = 99;`	1234	99	
`int t = a;`	1234	99	1234
`a = b;`	99	99	1234
`b = t;`	99	1234	1234

Your first trace

Expressions. An expression is a literal, a variable, or a sequence of operations on literals and/or variables that produces a value. For primitive types, expressions look just like mathematical formulas, which are based on familiar symbols or *operators* that specify data-type operations to be performed on one or more *operands*. Each operand can be any expression. Most of the operators that we use are *binary operators* that take exactly two operands, such as `x + 1` or `y / 2`. An expression that is enclosed in parentheses is another expression with the same value. For example, we can write `4 * (x - 3)` or `4*x - 12` on the right-hand side of an assignment statement and the compiler will understand what we mean.



Anatomy of an expression

Precedence. Such expressions are shorthand for specifying a sequence of computations: in what order should they be performed? Java has natural and well-defined *precedence* rules (see the booksite) that fully specify this order. For arithmetic operations, multiplication and division are performed before addition and subtraction, so that `a-b*c` and `a-(b*c)` represent the same sequence of operations. When arithmetic operators have the same precedence, the order is determined by *left-associativity*, so that `a-b-c` and `(a-b)-c` represent the same sequence of operations. You can use parentheses to override the rules, so you should not need to worry about the details of precedence for most of the programs that you write. (Some of the programs that you *read* might depend subtly on precedence rules, but we avoid such programs in this book.)

Converting strings to primitive values for command-line arguments. Java provides the library methods that we need to convert the strings that we type as command-line arguments into numeric values for primitive types. We use the Java library methods `Integer.parseInt()` and `Double.parseDouble()` for this purpose. For example, typing `Integer.parseInt("123")` in program text yields the literal value `123` (typing `123` has the same effect) and the code `Integer.parseInt(args[0])` produces the same result as the literal value typed as a string on the command line. You will see several examples of this usage in the programs in this section.

Converting primitive type values to strings for output. As mentioned at the beginning of this section, the Java built-in `String` type obeys special rules. One of these special rules is that you can easily convert any type of data to a `String`: whenever we use the + operator with a `String` as one of its operands, Java automatically converts the other to a `String`, producing as a result the `String` formed from the characters of the first operand followed by the characters of the second operand. For example, the result of these two code fragments

```
String a = "1234";            String a = "1234";
String b = "99";              int b = 99;
String c = a + b;             String c = a + b;
```

are both the same: they assign to `c` the value `"123499"`. We use this automatic conversion liberally to form `String` values for `System.out.print()` and `System.out.println()` for output. For example, we can write statements like this one:

```
System.out.println(a + " + " + b + " = " + c);
```

If `a`, `b`, and `c` are `int` variables with the values `1234`, `99`, and `1333`, respectively, then this statement prints out the string `1234 + 99 = 1333`.

With these mechanisms, our view of each Java program as a black box that takes string arguments and produces string results is still valid, but we can now interpret those strings as numbers and use them as the basis for meaningful computation. Next, we consider these details for the basic built-in types that you will use most often (strings, integers, floating-point numbers, and true–false values), along with sample code illustrating their use. To understand how to use a data type, you need to know not just its defined set of values, but also which operations you can perform, the language mechanism for invoking the operations, and the conventions for specifying literal values.

Characters and Strings A `char` is an alphanumeric character or symbol, like the ones that you type. There are 2^{16} different possible character values, but we usually restrict attention to the ones that represent letters, numbers, symbols, and whitespace characters such as tab and newline. Literals for `char` are characters enclosed in single quotes; for example, `'a'` represents the letter a. For tab, newline, backslash, single quote and double quote, we use the special *escape sequences* `'\t'`, `'\n'`, `'\\'`, `'\''`, and `'\"'`, respectively. The characters are encoded as 16-bit integers using an encoding scheme known as Unicode, and there are escape sequences for specifying special characters not found on your keyboard (see the booksite). We usually do not perform any operations directly on characters other than assigning values to variables.

values	sequences of characters
typical literals	`"Hello," "1 " " * "`
operation	concatenate
operator	+

Java's built-in `String` data type

A `String` is a sequence of characters. A literal `String` is a sequence of characters within double quotes, such as `"Hello, World"`. The `String` data type is *not* a primitive type, but Java sometimes treats it like one. For example, the *concatenation* operator (+) that we just considered is built in to the language as a binary operator in the same way as familiar operations on numbers.

The concatenation operation (along with the ability to declare `String` variables and to use them in expressions and assignment statements) is sufficiently powerful to allow us to attack some nontrivial computing tasks. As an example, `Ruler` (Program 1.2.1) computes a table of values of the *ruler function* that describes the relative lengths of the marks on a ruler. One noteworthy feature of this computation is that it illustrates how easy is is to craft short programs that produce huge amounts of output. If you extend this program in the obvious way to print five lines, six lines, seven lines, and so forth, you will see that each time you add just two statements to this program, you increase the size of its output by precisely one more than a factor of two. Specifically, if the program prints n lines, the nth line contains 2^n-1 numbers. For example, if you were to add statements in this way so that the program prints 30 lines, it would attempt to print more than 1 *billion* numbers.

expression	*value*
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

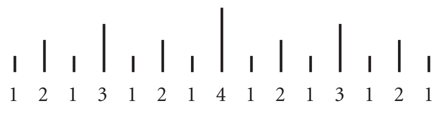
Typical `String` expressions

Program 1.2.1 String concatenation example

```
public class Ruler
{
   public static void main(String[] args)
   {
      String ruler1 = "1";
      String ruler2 = ruler1 + " 2 " + ruler1;
      String ruler3 = ruler2 + " 3 " + ruler2;
      String ruler4 = ruler3 + " 4 " + ruler3;
      System.out.println(ruler1);
      System.out.println(ruler2);
      System.out.println(ruler3);
      System.out.println(ruler4);
   }
}
```

This program prints the relative lengths of the subdivisions on a ruler. The nth line of output is the relative lengths of the marks on a ruler subdivided in intervals of 1/2ⁿ of an inch. For example, the fourth line of output gives the relative lengths of the marks that indicate intervals of one-sixteenth of an inch on a ruler.

```
% javac Ruler.java
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```



The ruler function for $n = 4$

As just discussed, our most frequent use (by far) of the concatenation operation is to put together results of computation for output with `System.out.print()` and `System.out.println()`. For example, we could simplify `UseArgument` (Program 1.1.2) by replacing its three statements with this single statement:

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

We have considered the `String` type first precisely because we need it for output (and command-line input) in programs that process other types of data.

Integers An `int` is an integer (natural number) between `-2147483648` (-2^{31}) and `2147483647` ($2^{31}-1$). These bounds derive from the fact that integers are represented in binary with 32 binary digits: there are 2^{32} possible values. (The term *binary digit* is omnipresent in computer science, and we nearly always use the abbreviation *bit*: a bit is either 0 or 1.) The range of possible `int` values is asymmetric because zero is included with the positive values. See the booksite for more details about number representation, but in the present context it suffices to know that an `int` is one of the finite set of values in the range just given. Sequences of the characters `0` through `9`, possibly with a plus or minus sign at the beginning (that, when interpreted as decimal numbers, fall within the defined range), are integer literal values. We use `int`s frequently because they naturally arise when implementing programs.

Standard arithmetic operators for addition/subtraction (`+` and `-`), multiplication (`*`), division (`/`), and remainder (`%`) for the `int` data type are built in to Java. These operators take two `int` operands and produce an `int` result, with one significant exception—division or remainder by zero is not allowed. These operations are defined just as in grade school (keeping in mind that all results must be integers): given two `int` values `a` and `b`, the value of `a / b` is the number of times `b` goes into `a` *with the fractional part discarded*, and the value of `a % b` is the remainder that you get when you divide `a` by `b`. For example, the value of `17 / 3` is `5`, and the value of `17 % 3` is `2`. The `int` results that we get from arithmetic operations are just what we expect, except that if the result is too large to fit into `int`'s 32-bit representation, then it will be truncated in a well-defined manner. This situation is known as *overflow*. In

expression	*value*	*comment*
`5 + 3`	`8`	
`5 - 3`	`2`	
`5 * 3`	`15`	
`5 / 3`	`1`	no fractional part
`5 % 3`	`2`	remainder
`1 / 0`		run-time error
`3 * 5 - 2`	`13`	`*` has precedence
`3 + 5 / 2`	`5`	`/` has precedence
`3 - 5 - 2`	`-4`	left associative
`( 3 - 5 ) - 2`	`-4`	better style
`3 - ( 5 - 2 )`	`0`	unambiguous

Typical `int` expressions

values	integers between -2^{31} and $+2^{31}-1$				
typical literals	`1234 99 -99 0 1000000`				
operations	add	subtract	multiply	divide	remainder
operators	`+`	`-`	`*`	`/`	`%`

Java's built-in `int` data type

Program 1.2.2 Integer multiplication and division

```
public class IntOps
{
   public static void main(String[] args)
   {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int p = a * b;
      int q = a / b;
      int r = a % b;
      System.out.println(a + " * " + b + " = " + p);
      System.out.println(a + " / " + b + " = " + q);
      System.out.println(a + " % " + b + " = " + r);
      System.out.println(a + " = " + q + " * " + b + " + " + r);
   }
}
```

Arithmetic for integers is built in to Java. Most of this code is devoted to the task of getting the values in and out; the actual arithmetic is in the simple statements in the middle of the program that assign values to `p`*,* `q`*, and* `r`*.*

```
% javac IntOps.java
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12 * 99 + 46
```

general, we have to take care that such a result is not misinterpreted by our code. For the moment, we will be computing with small numbers, so you do not have to worry about these boundary conditions.

PROGRAM 1.2.2 illustrates basic operations for manipulating integers, such as the use of expressions involving arithmetic operators. It also demonstrates the use of `Integer.parseInt()` to convert `String` values on the command line to `int` values, as well as the use of automatic type conversion to convert `int` values to `String` values for output.

Three other built-in types are different representations of integers in Java. The `long`, `short`, and `byte` types are the same as `int` except that they use 64, 16, and 8 bits respectively, so the range of allowed values is accordingly different. Programmers use `long` when working with huge integers, and the other types to save space. You can find a table with the maximum and minimum values for each type on the booksite, or you can figure them out for yourself from the numbers of bits.

Floating-point numbers The `double` type is for representing *floating-point* numbers, for use in scientific and commercial applications. The internal representation is like scientific notation, so that we can compute with numbers in a huge range. We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can only represent a finite number of floating-points in any digital computer representation. Floating-point numbers do approximate real numbers sufficiently well that we can use them in applications, but we often need to cope with the fact that we cannot always do exact computations.

expression	*value*
`3.141 + .03`	`3.171`
`3.141 - .03`	`3.111`
`6.02e23 / 2.0`	`3.01e23`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Typical `double` *expressions*

We can use a sequence of digits with a decimal point to type floating-point numbers. For example, `3.14159` represents a six-digit approximation to π. Alternatively, we can use a notation like scientific notation: the literal `6.022e23` represents the number 6.022×10^{23}. As with integers, you can use these conventions to write floating-point literals in your programs or to provide floating-point numbers as string parameters on the command line.

The arithmetic operators `+`, `-`, `*`, and `/` are defined for `double`. Beyond the built-in operators, the Java `Math` library defines the square root, trigonometric

values	real numbers (specified by IEEE 754 standard)			
typical literals	`3.14159   6.022e23   -3.0   2.0   1.4142135623730951`			
operations	add	subtract	multiply	divide
operators	`+`	`-`	`*`	`/`

Java's built-in `double` *data type*

Program 1.2.3 Quadratic formula

```
public class Quadratic
{
   public static void main(String[] args)
   {
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);
      double discriminant = b*b - 4.0*c;
      double d = Math.sqrt(discriminant);
      System.out.println((-b + d) / 2.0);
      System.out.println((-b - d) / 2.0);
   }
}
```

This program prints out the roots of the polynomial $x^2 + bx + c$, using the quadratic formula. For example, the roots of $x^2 - 3x + 2$ are 1 and 2 since we can factor the equation as $(x - 1)(x - 2)$; the roots of $x^2 - x - 1$ are ϕ and $1 - \phi$, where ϕ is the golden ratio, and the roots of $x^2 + x + 1$ are not real numbers.

```
% javac Quadratic.java
% java Quadratic -3.0 2.0
2.0
1.0
```

```
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949

% java Quadratic 1.0 1.0
NaN
NaN
```

functions, logarithm/exponential functions, and other common functions for floating-point numbers. To use one of these values in an expression, we write the name of the function followed by its argument in parentheses. For example, you can use the code `Math.sqrt(2.0)` when you want to use the square root of 2 in an expression. We discuss in more detail the mechanism behind this arrangement in Section 2.1 and more details about the `Math` library at the end of this section.

When working with floating point numbers, one of the first things that you will encounter is the issue of *precision*: `5.0/2.0` is `2.5` but `5.0/3.0` is `1.6666666666666667`. In Section 1.5, you will learn Java's mechanism for controlling the number of significant digits that you see in output. Until then, we will work with the Java default output format.

The result of a calculation can be one of the special values `Infinity` (if the number is too large to be represented) or `NaN` (if the result of the calculation is undefined). Though there are myriad details to consider when calculations involve these values, you can use `double` in a natural way and begin to write Java programs instead of using a calculator for all kinds of calculations. For example, PROGRAM 1.2.3 shows the use of `double` values in computing the roots of a quadratic equation using the quadratic formula. Several of the exercises at the end of this section further illustrate this point.

As with `long`, `short`, and `byte` for integers, there is another representation for real numbers called `float`. Programmers sometimes use `float` to save space when precision is a secondary consideration. The `double` type is useful for about 15 significant digits; the `float` type is good for only about 7 digits. We do not use `float` in this book.

values	true or false		
literals	`true` `false`		
operations	and	or	not
operators	`&&`	`||`	`!`

Java's built-in `boolean` *data type*

Booleans

The `boolean` type has just two values: `true` and `false`. These are the two possible `boolean` literals. Every `boolean` variable has one of these two values, and every `boolean` operation has operands and a result that takes on just one of these two values. This simplicity is deceiving—`boolean` values lie at the foundation of computer science.

The most important operations defined for `booleans` are *and* (`&&`), *or* (`||`), and *not* (`!`), which have familiar definitions:

- `a && b` is `true` if both operands are `true`, and `false` if either is `false`.
- `a || b` is `false` if both operands are `false`, and `true` if either is `true`.
- `!a` is `true` if `a` is `false`, and `false` if `a` is `true`.

Despite the intuitive nature of these definitions, it is worthwhile to fully specify each possibility for each operation in tables known as *truth tables*. The *not* function

`a`	`!a`
`true`	`false`
`false`	`true`

`a`	`b`	`a && b`	`a || b`
`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`
`true`	`false`	`false`	`true`
`true`	`true`	`true`	`true`

Truth-table definitions of `boolean` *operations*

`a`	`b`	`a && b`	`!a`	`!b`	`!a || !b`	`!(!a || !b)`
`false`	`false`	`false`	`true`	`true`	`true`	`false`
`false`	`true`	`false`	`true`	`false`	`true`	`false`
`true`	`false`	`false`	`false`	`true`	`true`	`false`
`true`	`true`	`true`	`false`	`false`	`false`	`true`

Truth-table proof that `a && b` *and* `!(!a || !b)` *are identical*

has only one operand: its value for each of the two possible values of the operand is specified in the second column. The *and* and *or* functions each have two operands: there are four different possibilities for operand input values, and the values of the functions for each possibility are specified in the right two columns.

We can use these operators with parentheses to develop arbitrarily complex expressions, each of which specifies a well-defined boolean function. Often the same function appears in different guises. For example, the expressions `(a && b)` and `!(!a || !b)` are equivalent.

The study of manipulating expressions of this kind is known as *Boolean logic*. This field of mathematics is fundamental to computing: it plays an essential role in the design and operation of computer hardware itself, and it is also a starting point for the theoretical foundations of computation. In the present context, we are interested in `boolean` expressions because we use them to control the behavior of our programs. Typically, a particular condition of interest is specified as a boolean expression and a piece of program code is written to execute one set of statements if the expression is `true` and a different set of statements if the expression is `false`. The mechanics of doing so are the topic of Section 1.3.

Comparisons Some *mixed-type* operators take operands of one type and produce a result of another type. The most important operators of this kind are the comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=`, which all are defined for each primitive numeric type and produce a `boolean` result. Since operations are defined only

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Typical comparison expressions

Program 1.2.4 Leap year

```
public class LeapYear
{
   public static void main(String[] args)
   {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;
      isLeapYear = (year % 4 == 0);
      isLeapYear = isLeapYear && (year % 100 != 0);
      isLeapYear = isLeapYear || (year % 400 == 0);
      System.out.println(isLeapYear);
   }
}
```

This program tests whether an integer corresponds to a leap year in the Gregorian calendar. A year is a leap year if it is divisible by 4 (2004), unless it is divisible by 100 in which case it is not (1900), unless it is divisible by 400 in which case it is (2000).

```
% javac LeapYear.java
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

with respect to data types, each of these symbols stands for many operations, one for each data type. It is required that both operands be of the same type. The result is always `boolean`.

Even without going into the details of number representation, it is clear that the operations for the various types are really quite different: for example, it is one thing to compare two `int`s to check that `(2 <= 2)` is `true` but quite another to compare two `double`s to check whether `(2.0 <= 0.002e3)` is `true` or `false`. Still, these operations are well-defined and useful to write code that tests for conditions such as `(b*b - 4.0*a*c) >= 0.0`, which is frequently needed, as you will see.

The comparison operations have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like `(b*b - 4.0*a*c) >= 0.0`, and you could write an expression like `month >= 1 && month <= 12` without parentheses to test whether the value of the `int` variable `month` is between 1 and 12. (It is better style to use the parentheses, however.)

op	*meaning*	`true`	`false`
`==`	*equal*	`2 == 2`	`2 == 3`
`!=`	*not equal*	`3 != 2`	`2 != 2`
`<`	*less than*	`2 < 13`	`2 < 2`
`<=`	*less than or equal*	`2 <= 2`	`3 <= 2`
`>`	*greater than*	`13 > 2`	`2 > 13`
`>=`	*greater than or equal*	`3 >= 2`	`2 >= 3`

Comparisons with `int` *operands and a* `boolean` *result*

Comparison operations, together with boolean logic, provide the basis for decision-making in Java programs. PROGRAM 1.2.4 is an example of their use, and you can find other examples in the exercises at the end of this section. More importantly, in SECTION 1.3 we will see the role that boolean expressions play in more sophisticated programs.

Library methods and APIs As we have seen, many programming tasks involve using Java library methods in addition to the built-in operations on data-type values. The number of available library methods is vast. As you learn to program, you will learn to use more and more library methods, but it is best at the beginning to restrict your attention to a relatively small set of methods. In this chapter, you have already used some of Java's methods for printing, for converting data from one type to another, and for computing mathematical functions (the Java `Math` library). In later chapters, you will learn not just how to use other methods, but how to create and use your own methods.

For convenience, we will consistently summarize the library methods that you need to know how to use in tables like this one:

`public class System.out`	
`void print(String s)`	*print* `s`
`void println(String s)`	*print* `s`*, followed by a newline*
`void println()`	*print a newline*

Note: Any type of data can be used (and will be automatically converted to `String`*).*

Excerpts from Java's library for standard output

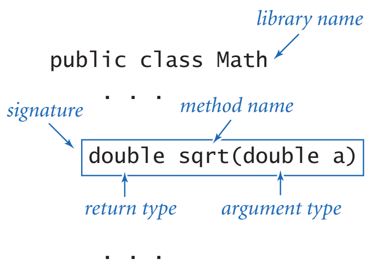
Such a table is known as an *application programming interface* (*API*). It provides the information that you need to write an *application program* that uses the methods. Here is an API for the most commonly used methods in Java's `Math` library:

`public class Math`	
`double abs(double a)`	*absolute value of a*
`double max(double a, double b)`	*maximum of a and b*
`double min(double a, double b)`	*minimum of a and b*
Note 1: `abs()`, `max()`, *and* `min()` *are defined also for* `int`, `long`, *and* `float`.	
`double sin(double theta)`	*sine function*
`double cos(double theta)`	*cosine function*
`double tan(double theta)`	*tangent function*
Note 2: Angles are expressed in radians. Use `toDegrees()` *and* `toRadians()` *to convert.* *Note 3: Use* `asin()`, `acos()`, *and* `atan()` *for inverse functions.*	
`double exp(double a)`	*exponential* (e^a)
`double log(double a)`	*natural log* ($\log_e a$, *or* $\ln a$)
`double pow(double a, double b)`	*raise a to the bth power* (a^b)
`long round(double a)`	*round to the nearest integer*
`double random()`	*random number in* [0, 1)
`double sqrt(double a)`	*square root of a*
`double E`	*value of e (constant)*
`double PI`	*value of* π *(constant)*

See booksite for other available functions.

Excerpts from Java's mathematics library

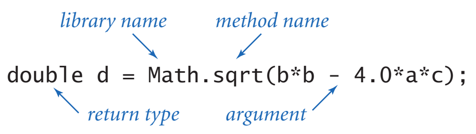
With the exception of `random()`, these methods implement mathematical functions—they use their arguments to compute a value of a specified type. Each method is described by a line in the API that specifies the information you need to know in order to use the method. The code in the tables is *not* the code that you type to use the method; it is known as the method's *signature*. The signature specifies the type of the arguments, the method name, and the type of the value that the method computes (the *return value*). When your program is executed, we say that it *calls* the system library code for the method, which *returns* the value for use in your code.



Anatomy of a method signature

Note that `random()` does not implement a mathematical function because it does not take an argument. On the other hand, `System.out.print()` and `System.out.println()` do not implement mathematical functions because they do not return values and therefore do not have a return type. (This condition is specified in the signature by the keyword `void`.)

In your code, you can use a library method by typing its name followed by arguments of the specified type, enclosed in parentheses and separated by commas. You can use this code in the same way as you use variables and literals in expressions. When you do so, you can expect that method to compute a value of the appropriate type, as documented in the left column of the API. For example, you can write expressions like `Math.sin(x) * Math.cos(y)` and so on. Method arguments may also be expressions, as in `Math.sqrt(b*b - 4.0*a*c)`.



Using a library method

The `Math` library also defines the precise constant values `PI` (for π) and `E` (for *e*), so that you can use those names to refer to those constants in your programs. For example, the value of `Math.sin(Math.PI/2)` is `1.0` and the value of `Math.log(Math.E)` is `1.0` (because `Math.sin()` takes its argument in radians and `Math.log()` implements the natural logarithm function).

To be complete, we also include here the following API for Java's conversion methods, which we use for command-line arguments:

`int`	`Integer.parseInt(String s)`	*convert* `s` *to an* `int` *value*
`double`	`Double.parseDouble(String s)`	*convert* `s` *to a* `double` *value*
`long`	`Long.parseLong(String s)`	*convert* `s` *to a* `long` *value*

Java library methods for converting strings to primitive types

You *do not* need to use methods like these to convert from `int`, `double`, and `long` values to `String` values for *output*, because Java automatically converts any value used as an argument to `System.out.print()` or `System.out.println()` to `String` for output.

expression	*library*	*type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.random()`	`Math`	`double`	*random in* [0, 1)
`Math.round(3.14159)`	`Math`	`long`	`3`

Typical expressions that use Java library methods

These APIs are typical of the online documentation that is the standard in modern programming. There is extensive online documentation of the Java APIs that is used by professional programmers, and it is available to you (if you are interested) directly from the Java website or through our booksite. You do not need to go to the online documentation to understand the code in this book or to write similar code, because we present and explain in the text all of the library methods that we use in APIs like these and summarize them in the endpapers. More important, in Chapters 2 and 3 you will learn in this book how to develop your own APIs and to implement functions for your own use.

Type conversion One of the primary rules of modern programming is that you should always be aware of the type of data that your program is processing. Only by knowing the type can you know precisely which set of values each variable can have, which literals you can use, and which operations you can perform. Typical programming tasks involve processing multiple types of data, so we often need to convert data from one type to another. There are several ways to do so in Java.

Explicit type conversion. You can use a method that takes an argument of one type (the value to be converted) and produces a result of another type. We have already used the `Integer.parseInt()` and `Double.parseDouble()` library methods to convert `String` values to `int` and `double` values, respectively. Many other methods are available for conversion among other types. For example, the library method `Math.round()` takes a `double` argument and returns a `long` result: the nearest integer to the argument. Thus, for example, `Math.round(3.14159)` and `Math.round(2.71828)` are both of type `long` and have the same value (`3`).

Explicit cast. Java has some built-in type conversion conventions for primitive types that you can take advantage of when you are aware that you might lose information. You have to make your intention to do so explicit by using a device called a *cast*. You cast an expression from one primitive type to another by prepending the desired type name within parentheses. For example, the expression `(int) 2.71828` is a cast from `double` to `int` that produces an `int` with value `2`. The conversion methods defined for casts throw away information in a reasonable way (for a full list, see the booksite). For example, casting a floating-point number to an integer discards the fractional part by rounding towards zero. If you want a different result, such as rounding to the nearest integer, you must use the explicit conversion method `Math.round()`, as just discussed (but you then need to use an explicit cast to `int`, since that method returns a `long`). `RandomInt` (Program 1.2.5) is an example that uses a cast for a practical computation.

Automatic promotion for numbers. You can use data of any primitive numeric type where a value whose type has a larger range of values is expected, because Java automatically converts to the type with the larger range. This kind of conversion is called *promotion*. For example, we used numbers all of type `double` in Program 1.2.3, so there is no conversion. If we had chosen to make `b` and `c` of type `int` (using `Integer.parseInt()` to convert the command-line arguments), automatic promotion would be used to evaluate the expression `b*b - 4.0*c`. First, `c` is promoted to `double` to multiply by the `double` literal `4.0`, with a `double` result. Then, the `int` value `b*b` is promoted to `double` for the subtraction, leaving a `double` result. Or, we might have written `b*b - 4*c`. In that case, the expression `b*b - 4*c` would be evaluated as an `int` and then the result promoted to `double`, because that is what `Math.sqrt()` expects. Promotion is appropriate because your intent is clear and it can be done with no loss of information. On the other hand, a conversion that might involve loss of information (for example, assigning a `double` to an `int`) leads to a compile-time error.

expression	*expression type*	*expression value*
`"1234" + 99`	`String`	`"123499"`
`Integer.parseInt("123")`	`int`	`123`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`(int) Math.round(3.14159)`	`int`	`3`
`11 * 0.3`	`double`	`3.3`
`(int) 11 * 0.3`	`double`	`3.3`
`11 * (int) 0.3`	`int`	`0`
`(int) (11 * 0.3)`	`int`	`3`

Typical type conversions

Program 1.2.5 Casting to get a random integer

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      double r = Math.random();  // uniform between 0 and 1
      int n = (int) (r * N);     // uniform between 0 and N-1
      System.out.println(n);
   }
}
```

This program uses the Java method `Math.random()` *to generate a random number* `r` *in the interval* [0, 1)*, then multiplies* `r` *by the command-line argument* `N` *to get a random number greater than or equal to* `0` *and less than* `N`*, then uses a cast to truncate the result to be an integer* `n` *between* `0` *and* `N-1`*.*

```
% javac RandomInt.java

% java RandomInt 1000
548

% java RandomInt 1000
141

% java RandomInt 1000000
135032
```

Casting has higher precedence than arithmetic operations—any cast is applied to the value that immediately follows it. For example, if we write `int n = (int) 11 * 0.3`, the cast is no help: the literal `11` is already an integer, so the cast `(int)` has no effect. In this example, the compiler produces a `possible loss of precision` error message because there would be a loss of precision in converting the resulting value (`3.3`) to an `int` for assignment to `n`. The error is helpful because the intended computation for this code is likely `(int) (11 * 0.3)`, which has the value `3`, not `3.3`.

Beginning programmers tend to find type conversion to be an annoyance, but experienced programmers know that paying careful attention to data types is a key to success in programming. It is well worth your while to take the time to understand what type conversion is all about. After you have written just a few programs, you will understand that these rules help you to make your intentions explicit and to avoid subtle bugs in your programs.

Summary *A data type is a set of values and a set of operations on those values.* Java has eight primitive data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In Java code, we use operators and expressions like those in familiar mathematical expressions to invoke the operations associated with each type. The `boolean` type is for computing with the logical values `true` and `false`; the `char` type is the set of character values that we type; and the other six are numeric types, for computing with numbers. In this book, we most often use `boolean`, `int`, and `double`; we do not use `short` or `float`. Another data type that we use frequently, `String`, is not primitive, but Java has some built-in facilities for `String`s that are like those for primitive types.

When programming in Java, we have to be aware that every operation is defined only in the context of its data type (so we may need type conversions) and that all types can have only a finite number of values (so we may need to live with imprecise results).

The `boolean` type and its operations— `&&`, `||`, and `!` —are the basis for logical decision-making in Java programs, when used in conjunction with the mixed-type comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`. Specifically, we use `boolean` expressions to control Java's conditional (`if`) and loop (`for` and `while`) constructs, which we will study in detail in the next section.

The numeric types and Java's libraries give us the ability to use Java as an extensive mathematical calculator. We write arithmetic expressions using the built-in operators `+`, `-`, `*`, `/`, and `%` along with Java methods from the `Math` library. Although the programs in this section are quite rudimentary by the standards of what we will be able to do after the next section, this class of programs is quite useful in its own right. You will use primitive types and basic mathematical functions extensively in Java programming, so the effort that you spend now understanding them will certainly be worthwhile.

Q&A

Q. What happens if I forget to declare a variable?

A. The compiler complains, as shown below for a program `IntOpsBad`, which is the same as Program 1.2.2 except that the `int` variable `p` is omitted from the declaration statement.

```
% javac IntOpsBad.java
IntOpsBad.java:7: cannot resolve symbol
symbol : variable p
location: class IntOpsBad
p = a * b;
           ^
IntOpsBad.java:10: cannot resolve symbol
symbol : variable p
location: class IntOpsBad
System.out.println(a + " * " + b + " = " + p);
                                           ^
2 errors
```

The compiler says that there are two errors, but there is really just one: the declaration of `p` is missing. If you forget to declare a variable that you use often, you will get quite a few error messages. A good strategy is to correct the *first* error and check that correction before addressing later ones.

Q. What happens if I forget to initialize a variable?

A. The compiler checks for this condition and will give you a `variable might not have been initialized` error message if you try to use the variable in an expression.

Q. Is there a difference between = and == ?

A. Yes, they are quite different! The first is an assignment operator that changes the value of a variable, and the second is a comparison operator that produces a `boolean` result. Your ability to understand this answer is a sure test of whether you understood the material in this section. Think about how you might explain the difference to a friend.

Q. Why do `int` values sometime become negative when they get large?

A. If you have not experienced this phenomenon, see Exercise 1.2.10. The problem has to do with the way integers are represented in the computer. You can learn the details on the booksite. In the meantime, a safe strategy is using the `int` type when you know the values to be less than ten digits and the `long` type when you think the values might get to be ten digits or more.

Q. It seems wrong that Java should just let `int`s overflow and give bad values. Shouldn't Java automatically check for overflow?

A. Yes, this issue is a contentious one among programmers. The short answer for now is that the lack of such checking is one reason such types are called *primitive* data types. A little knowledge can go a long way in avoiding such problems. Again, it is fine to use the `int` type for small numbers, but when values run into the billions, you cannot.

Q. What is the value of `Math.abs(-2147483648)`?

A. `-2147483648`. This strange (but true) result is a typical example of the effects of integer overflow.

Q. It is annoying to see all those digits when printing a `float` or a `double`. Can we get `System.out.println()` to print out just two or three digits after the decimal point?

A. That sort of task involves a closer look at the method used to convert from `double` to `String`. The Java library function `System.out.printf()` is one way to do the job, and it is similar to the basic printing method in the C programming language and many modern languages, as discussed in Section 1.5. Until then, we will live with the extra digits (which is not all bad, since doing so helps us to get used to the different primitive types of numbers).

Q. How can I initialize a `double` variable to infinity?

A. Java has built-in constants available for this purpose: `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`.

Q. What is the value of `Math.round(6.022e23)`?

A. You should get in the habit of typing in a tiny Java program to answer such questions yourself (and trying to understand why your program produces the result that it does).

Q. Can you compare a `double` to an `int`?

A. Not without doing a type conversion, but remember that Java usually does the requisite type conversion automatically. For example, if `x` is an `int` with the value `3`, then the expression `(x < 3.1)` is `true`—Java converts `x` to `double` (because `3.1` is a `double` literal) before performing the comparison.

Q. Are expressions like `1/0` and `1.0/0.0` legal in Java?

A. No and yes. The first generates a run-time *exception* for division by zero (which stops your program because the value is undefined); the second is legal and has the value `Infinity`.

Q. Are there functions in Java's `Math` library for other trigonometric functions, like cosecant, secant, and cotangent?

A. No, because you could use `Math.sin()`, `Math.cos()`, and `Math.tan()` to compute them. Choosing which functions to include in an API is a tradeoff between the convenience of having every function that you need and the annoyance of having to find one of the few that you need in a long list. No choice will satisfy all users, and the Java designers have many users to satisfy. Note that there are plenty of redundancies even in the APIs that we have listed. For example, you could use `Math.sin(x)/Math.cos(x)` instead of `Math.tan(x)`.

Q. Can you use < and > to compare `String` variables?

A. No. Those operators are defined only for primitive types.

Q. How about == and `!=` ?

A. Yes, but the result may not be what you expect, because of the meanings these operators have for non-primitive types. For example, there is a distinction between a `String` and its value. The expression `"abc" == "ab" + x` is `false` when `x` is a `String` with value `"c"` because the two operands are stored in different places in memory (even though they have the same value). This distinction is essential, as you will learn when we discuss it in more detail in SECTION 3.1.

Q. What is the result of division and remainder for negative integers?

A. The quotient `a / b` rounds toward `0`; the remainder `a % b` is defined such that `(a / b) * b + a % b` is always equal to `a`. For example, `-14/3` and `14/-3` are both `-4`, but `-14 % 3` is `-2` and `14 % -3` is `2`.

Q. Will `(a < b < c)` test whether three numbers are in order?

A. No, that will not compile. You need to say `(a < b && b < c)`.

Q. Fifteen digits for floating-point numbers certainly seems enough to me. Do I really need to worry much about precision?

A. Yes, because you are used to mathematics based on real numbers with infinite precision, whereas the computer always deals with approximations. For example, `(0.1 + 0.1 == 0.2)` is `true` but `(0.1 + 0.1 + 0.1 == 0.3)` is `false`! Pitfalls like this are not at all unusual in scientific computing. Novice programmers should avoid comparing two floating-point numbers for equality.

Q. Why do we say `(a && b)` and not `(a & b)`?

A. Java also has a `&` operator that we do not use in this book but which you may encounter if you pursue advanced programming courses.

Q. Why is the value of `10^6` not `1000000` but `12`?

A. The `^` operator is not an exponentiation operator, which you must have been thinking. Instead, it is an operator like `&` that we do not use in this book. You want the literal `1e6`. You could also use `Math.pow(10, 6)` but doing so is wasteful if you are raising `10` to a known power.

Exercises

1.2.1 Suppose that `a` and `b` are `int` values. What does the following sequence of statements do?

```
int t = a; b = t; a = b;
```

1.2.2 Write a program that uses `Math.sin()` and `Math.cos()` to check that the value of $\cos^2\theta + \sin^2\theta$ is approximately 1 for any θ entered as a command-line argument. Just print the value. Why are the values not always exactly 1?

1.2.3 Suppose that `a` and `b` are `int` values. Show that the expression

```
(!(a && b) && (a || b)) || ((a && b) || !(a || b))
```

is equivalent to `true`.

1.2.4 Suppose that `a` and `b` are `int` values. Simplify the following expression: `(!(a < b) && !(a > b))`.

1.2.5 The *exclusive or* operator `^` for `boolean` operands is defined to be `true` if they are different, `false` if they are the same. Give a truth table for this function.

1.2.6 Why does `10/3` give `3` and not `3.333333333`?

Solution. Since both `10` and `3` are integer literals, Java sees no need for type conversion and uses integer division. You should write `10.0/3.0` if you mean the numbers to be `double` literals. If you write `10/3.0` or `10.0/3`, Java does implicit conversion to get the same result.

1.2.7 What do each of the following print?

a. `System.out.println(2 + "bc");`

b. `System.out.println(2 + 3 + "bc");`

c. `System.out.println((2+3) + "bc");`

d. `System.out.println("bc" + (2+3));`

e. `System.out.println("bc" + 2 + 3);`

Explain each outcome.

1.2.8 Explain how to use PROGRAM 1.2.3 to find the square root of a number.

1.2.9 What do each of the following print?

a. `System.out.println('b');`

b. `System.out.println('b' + 'c');`

c. `System.out.println((char) ('a' + 4));`

Explain each outcome.

1.2.10 Suppose that a variable `a` is declared as `int a = 2147483647` (or equivalently, `Integer.MAX_VALUE`). What do each of the following print?

a. `System.out.println(a);`

b. `System.out.println(a+1);`

c. `System.out.println(2-a);`

d. `System.out.println(-2-a);`

e. `System.out.println(2*a);`

f. `System.out.println(4*a);`

Explain each outcome.

1.2.11 Suppose that a variable `a` is declared as `double a = 3.14159`. What do each of the following print?

a. `System.out.println(a);`

b. `System.out.println(a+1);`

c. `System.out.println(8/(int) a);`

d. `System.out.println(8/a);`

e. `System.out.println((int) (8/a));`

Explain each outcome.

1.2.12 Describe what happens if you write `sqrt` instead of `Math.sqrt` in PROGRAM 1.2.3.

1.2.13 What is the value of `(Math.sqrt(2) * Math.sqrt(2) == 2)` ?

1.2.14 Write a program that takes two positive integers as command-line arguments and prints `true` if either evenly divides the other.

1.2.15 Write a program that takes three positive integers as command-line arguments and prints `true` if any one of them is greater than or equal to the sum of the other two and `false` otherwise. (*Note*: This computation tests whether the three numbers could be the lengths of the sides of some triangle.)

1.2.16 A physics student gets unexpected results when using the code

```
F = G * mass1 * mass2 / r * r;
```

to compute values according to the formula $F = Gm_1m_2 / r^2$. Explain the problem and correct the code.

1.2.17 Give the value of `a` after the execution of each of the following sequences:

```
int a = 1;          boolean a = true;      int a = 2;
a = a + a;          a = !a;                a = a * a;
a = a + a;          a = !a;                a = a * a;
a = a + a;          a = !a;                a = a * a;
```

1.2.18 Suppose that `x` and `y` are `double` values that represent the Cartesian coordinates of a point (x, y) in the plane. Give an expression whose value is the distance of the point from the origin.

1.2.19 Write a program that takes two `int` values `a` and `b` from the command line and prints a random integer between `a` and `b`.

1.2.20 Write a program that prints the sum of two random integers between `1` and `6` (such as you might get when rolling dice).

1.2.21 Write a program that takes a `double` value `t` from the command line and prints the value of $\sin(2t) + \sin(3t)$.

1.2.22 Write a program that takes three `double` values x_0, v_0, and t from the command line and prints the value of $x_0 + v_0t + gt^2/2$, where g is the constant 9.78033. (*Note*: This value the displacement in meters after t seconds when an object is thrown straight up from initial position x_0 at velocity v_0 meters per second.)

1.2.23 Write a program that takes two `int` values `m` and `d` from the command line and prints `true` if day d of month m is between 3/20 and 6/20, `false` otherwise.

Creative Exercises

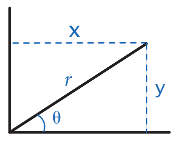
1.2.24 *Loan payments.* Write a program that calculates the monthly payments you would have to make over a given number of years to pay off a loan at a given interest rate compounded continuously, taking the number of years t, the principal P, and the annual interest rate r as command-line arguments. The desired value is given by the formula Pe^{rt}. Use `Math.exp()`.

1.2.25 *Wind chill.* Given the temperature t (in Fahrenheit) and the wind speed v (in miles per hour), the National Weather Service defines the effective temperature (the wind chill) to be:

$$w = 35.74 + 0.6215\,t + (0.4275\,t - 35.75)\,v^{0.16}$$

Write a program that takes two `double` command-line arguments `t` and `v` and prints out the wind chill. Use `Math.pow(a, b)` to compute a^b. *Note*: The formula is not valid if t is larger than 50 in absolute value or if v is larger than 120 or less than 3 (you may assume that the values you get are in that range).

1.2.26 *Polar coordinates.* Write a program that converts from Cartesian to polar coordinates. Your program should take two real numbers `x` and `y` on the command line and print the polar coordinates r and θ. Use the Java method `Math.atan2(y, x)` which computes the arctangent value of `y/x` that is in the range from $-\pi$ to π.



Polar coordinates

1.2.27 *Gaussian random numbers.* One way to generate a random number taken from the Gaussian distribution is to use the *Box-Muller* formula

$$w = \sin(2\,\pi\,v)\,(-2\ln u)^{1/2}$$

where u and v are real numbers between 0 and 1 generated by the `Math.random()` method. Write a program `StdGaussian` that prints out a standard Gaussian random variable.

1.2.28 *Order check.* Write a program that takes three `double` values `x`, `y`, and `z` as command-line arguments and prints `true` if the values are strictly ascending or descending ($x < y < z$ or $x > y > z$), and `false` otherwise.

1.2.29 *Day of the week.* Write a program that takes a date as input and prints the day of the week that date falls on. Your program should take three command line parameters: `m` (month), `d` (day), and `y` (year). For `m`, use `1` for January, `2` for February, and so forth. For output, print `0` for Sunday, `1` for Monday, `2` for Tuesday, and so forth. Use the following formulas, for the Gregorian calendar:

$$
\begin{aligned}
y_0 &= y - (14 - m) / 12 \\
x &= y_0 + y_0/4 - y_0/100 + y_0/400 \\
m_0 &= m + 12 \times ((14 - m) / 12) - 2 \\
d_0 &= (d + x + (31 \times m_0)/12) \ \% \ 7
\end{aligned}
$$

Example: On what day of the week was February 14, 2000?

$$
\begin{aligned}
y_0 &= 2000 - 1 = 1999 \\
x &= 1999 + 1999/4 - 1999/100 + 1999/400 = 2483 \\
m_0 &= 2 + 12 \times 1 - 2 = 12 \\
d_0 &= (14 + 2483 + (31 \times 12) / 12) \ \% \ 7 = 2500 \ \% \ 7 = 1
\end{aligned}
$$

Answer: Monday.

1.2.30 *Uniform random numbers.* Write a program that prints five uniform random values between 0 and 1, their average value, and their minimum and maximum value. Use `Math.random()`, `Math.min()`, and `Math.max()`.

1.2.31 *Mercator projection.* The *Mercator projection* is a conformal (angle preserving) projection that maps latitude φ and longitude λ to rectangular coordinates (x, y). It is widely used—for example, in nautical charts and in the maps that you print from the web. The projection is defined by the equations $x = \lambda - \lambda_0$ and $y = 1/2 \ln((1 + \sin\varphi) / (1 - \sin\varphi))$, where λ_0 is the longitude of the point in the center of the map. Write a program that takes λ_0 and the latitude and longitude of a point from the command line and prints its projection.

1.2.32 *Color conversion.* Several different formats are used to represent color. For example, the primary format for LCD displays, digital cameras, and web pages, known as the *RGB format,* specifies the level of red (R), green (G), and blue (B) on an integer scale from 0 to 255. The primary format for publishing books and magazines, known as the *CMYK format*, specifies the level of cyan (C), magenta (M), yellow (Y), and black (K) on a real scale from 0.0 to 1.0. Write a program `RGBtoCMYK` that converts RGB to CMYK. Take three integers—`r`, `g`, and `b`—from the command line and print the equivalent CMYK values. If the RGB values are all 0, then the CMY values are all 0 and the K value is 1; otherwise, use these formulas:

$$\begin{aligned} w &= \max(\, r/255,\, g/255,\, b/255 \,) \\ c &= (w - (r/255))/w \\ m &= (w - (g/255))/w \\ y &= (w - (b/255))/w \\ k &= 1 - w \end{aligned}$$

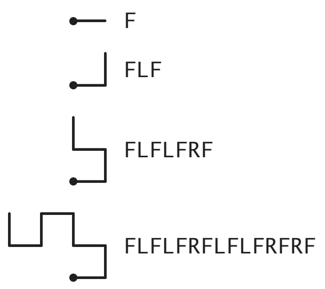
1.2.33 *Great circle.* Write a program `GreatCircle` that takes four command-line arguments—`x1`, `y1`, `x2`, and `y2`—(the latitude and longitude, in degrees, of two points on the earth) and prints out the great-circle distance between them. The great-circle distance (in nautical miles) is given by the equation:

$$d = 60 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2))$$

Note that this equation uses degrees, whereas Java's trigonometric functions use radians. Use `Math.toRadians()` and `Math.toDegrees()` to convert between the two. Use your program to compute the great-circle distance between Paris (48.87° N and −2.33° W) and San Francisco (37.8° N and 122.4° W).

1.2.34 *Three-sort.* Write a program that takes three `int` values from the command line and prints them in ascending order. Use `Math.min() and Math.max()`.

1.2.35 *Dragon curves.* Write a program to print the instructions for drawing the dragon curves of order 0 through 5. The instructions are strings of `F`, `L`, and `R` characters, where `F` means "draw line while moving 1 unit forward," `L` means "turn left," and `R` means "turn right." A dragon curve of order N is formed when you fold a strip of paper in half N times, then unfold to right angles. The key to solving this problem is to note that a curve of order N is a curve of order $N-1$ followed by an `L` followed by a curve of order $N-1$ traversed in reverse order, and then to figure out a similar description for the reverse curve .



Dragon curves of order 0, 1, 2, and 3