

**A Java Reference:  
Assorted Java Reference Material**

Paul N. Hilfinger  
University of California, Berkeley

Copyright © 2001, 2002, 2004, 2006, 2008, 2011, 2012, 2013, 2014, 2018 by Paul N. Hilfinger. All rights reserved.

**Acknowledgments.** Many thanks to Soheil Golshan, Ashwin Iyengar, Kegan Kawamura, Benson Limketkai, Vadim Perelman, Barath Raghavan, Chen Tang, Michael Yang, and Asa Daniel Zernik for finding many errors in previous editions of this document. Remaining errors are, of course, my own.

# Contents

<b>1</b>	<b>Java Overview</b>	<b>9</b>
1.1	Basic Program Structure . . . . .	9
1.2	Compilation and Execution . . . . .	11
1.3	Simple Values and Expressions . . . . .	12
1.3.1	Writing Numbers . . . . .	12
1.3.2	Arithmetic . . . . .	13
1.3.3	Comparisons and Logical Operations . . . . .	14
1.3.4	Strings . . . . .	16
1.3.5	Static Methods: Abstracting Computation . . . . .	17
1.4	Conditional Execution . . . . .	18
1.4.1	If statements . . . . .	19
1.4.2	Conditional Expressions . . . . .	19
1.4.3	Case analysis and the Switch Statement . . . . .	19
1.5	Arrays I: The Command Line . . . . .	21
1.6	Example: Finding Primes . . . . .	22
1.6.1	Starting from the top . . . . .	23
1.6.2	Starting from the bottom . . . . .	23
1.6.3	Meeting in the middle . . . . .	26
1.7	Example: Pig Latin . . . . .	28
1.7.1	Vowels . . . . .	28
1.7.2	Translation . . . . .	29
1.7.3	Counting Consonants . . . . .	29
1.7.4	To the top . . . . .	31
1.8	Variables and Assignment . . . . .	32
1.9	Repetition . . . . .	34
1.9.1	Indefinite iteration . . . . .	35
1.9.2	Definite Iteration . . . . .	37
1.9.3	Example: Iterative Prime-Finding . . . . .	40
1.9.4	Example: Counting . . . . .	41
1.10	Arrays II: Creation . . . . .	42
1.10.1	Example: Linear Interpolation . . . . .	43
1.10.2	Example: The Sieve of Eratosthenes . . . . .	44
1.10.3	Multi-dimensional Arrays . . . . .	46
1.11	Introduction to Objects . . . . .	48

1.11.1	Simple Object Type Definition and Creation . . . . .	48
1.11.2	Instance Methods . . . . .	50
1.11.3	Constructors . . . . .	51
1.11.4	Example: A Prime-Number Class . . . . .	52
1.12	Interfaces . . . . .	53
1.13	Inheritance . . . . .	56
1.14	Packages and Access Control . . . . .	60
1.15	Handling Exceptional Cases . . . . .	62
1.15.1	Built-in Exceptions . . . . .	62
1.15.2	What Exactly is an Exception? . . . . .	62
1.15.3	Throwing Exceptions Explicitly . . . . .	63
1.15.4	Catching Exceptions . . . . .	63
<b>2</b>	<b>Describing a Programming Language</b>	<b>65</b>
2.1	Dynamic and Static Properties . . . . .	65
2.2	Describing Lexical Structure and Syntax . . . . .	67
<b>3</b>	<b>Lexical Basics</b>	<b>69</b>
3.1	Layout: Whitespace and Comments . . . . .	69
<b>4</b>	<b>Values, Types, and Containers</b>	<b>71</b>
4.1	Values and Containers . . . . .	72
4.1.1	Containers and Names . . . . .	72
4.1.2	Pointers . . . . .	72
4.2	Types . . . . .	74
4.2.1	Static vs. dynamic types . . . . .	75
4.2.2	Type denotations in Java . . . . .	76
4.3	Environments . . . . .	77
4.4	Applying the model to Java . . . . .	77
<b>5</b>	<b>Declarations</b>	<b>79</b>
5.1	Scope and the Meaning of Names . . . . .	80
5.1.1	Block structure . . . . .	80
5.1.2	Selection . . . . .	82
5.1.3	Packages and Imports . . . . .	82
5.2	Local Variables . . . . .	85
5.3	Type Declarations . . . . .	87
5.4	Class Declarations . . . . .	87
5.4.1	Kinds of class members . . . . .	88
5.4.2	Inheritance . . . . .	89
5.4.3	Class Modifiers . . . . .	89
5.4.4	Fields . . . . .	90
5.5	Interface Declarations . . . . .	91
5.6	Enumerated Types . . . . .	92
5.7	Nested Classes and Interfaces . . . . .	94
5.7.1	Member types . . . . .	94

5.7.2	Local classes . . . . .	95
5.7.3	Anonymous classes . . . . .	97
5.7.4	What on earth is this all for? . . . . .	98
5.8	Method (Function) Declarations . . . . .	99
5.8.1	Variable-Length Parameter Lists . . . . .	100
5.8.2	Method signatures . . . . .	101
5.8.3	Overloading and hiding methods . . . . .	103
5.8.4	Overriding . . . . .	104
5.9	Constructor Declarations . . . . .	107
5.9.1	Rationale. . . . .	108
5.9.2	Instance and field initializers . . . . .	109
5.9.3	Some Useful Constructor Idioms . . . . .	110
5.10	Initialization of Classes . . . . .	112
5.11	Access control . . . . .	113
5.12	The Java Program . . . . .	114
5.12.1	Executing a Program . . . . .	114
5.13	Annotations . . . . .	116
5.13.1	Deprecated . . . . .	117
5.13.2	Override . . . . .	117
5.13.3	SuppressWarnings . . . . .	118
<b>6</b>	<b>The Expression Language</b>	<b>119</b>
6.1	Syntactic overview . . . . .	119
6.1.1	Prefix, Postfix, and Infix Operators . . . . .	121
6.1.2	Literals . . . . .	122
6.2	Conversions and Casts . . . . .	123
6.2.1	Explicit conversions . . . . .	123
6.2.2	Implicit conversions . . . . .	124
6.2.3	Promotions . . . . .	125
6.3	Integers and Characters . . . . .	125
6.3.1	Integral values and their literals . . . . .	126
6.3.2	Modular integer arithmetic . . . . .	129
6.3.3	Manipulating bits . . . . .	132
6.4	Floating-Point Numbers . . . . .	135
6.4.1	Floating-Point Literals . . . . .	136
6.4.2	Floating-point arithmetic . . . . .	137
6.5	Booleans and Conditionals . . . . .	141
6.5.1	Boolean literals . . . . .	142
6.5.2	Boolean operations . . . . .	142
6.5.3	Comparisons and Equality . . . . .	142
6.5.4	Conditional expressions . . . . .	143
6.6	Reference Types . . . . .	143
6.6.1	Allocating class instances . . . . .	144
6.6.2	Field Access . . . . .	144
6.6.3	Testing types . . . . .	147

6.6.4	Arrays . . . . .	148
6.6.5	Strings . . . . .	151
6.7	Assignment, Increment, and Decrement Operators . . . . .	154
6.8	Method Calls . . . . .	156
6.8.1	Static method calls . . . . .	157
6.8.2	Instance Method Calls . . . . .	158
6.9	Constant Expressions . . . . .	160
6.10	Reflection . . . . .	160
6.10.1	Type java.lang.Class . . . . .	161
6.11	Definite Assignment and Related Obscure Details . . . . .	163
6.11.1	Assignments to fields . . . . .	164
6.11.2	Blank final variables . . . . .	165
<b>7</b>	<b>Statements</b>	<b>167</b>
7.1	Sequencing, Blocks, and Empty Statements . . . . .	168
7.2	Conditional Control . . . . .	169
7.2.1	If Statements . . . . .	169
7.2.2	Switch Statements . . . . .	170
7.3	About Statement Format . . . . .	173
7.4	Iteration . . . . .	174
7.4.1	While Loops . . . . .	174
7.4.2	Traditional For Loops . . . . .	175
7.4.3	For Loops for Collections and Arrays . . . . .	178
7.5	Jump Statements and Labels . . . . .	179
7.5.1	The ‘break’ Statement . . . . .	180
7.5.2	The ‘continue’ Statement . . . . .	182
7.5.3	The ‘return’ Statement . . . . .	183
<b>8</b>	<b>Exception Handling</b>	<b>185</b>
8.1	Throw Statements and Exception Types . . . . .	186
8.2	Catching Exceptions: Try Statements . . . . .	186
8.2.1	Simple example: single handler . . . . .	188
8.2.2	Multiple handlers. . . . .	189
8.2.3	Using the Caught Exception. . . . .	189
8.2.4	The Finally Clause . . . . .	190
8.3	Declaring Thrown Exceptions . . . . .	191
8.4	Exceptions in the java.lang Package . . . . .	194
<b>9</b>	<b>Strings, Streams, and Patterns</b>	<b>197</b>
9.1	Bytes and Characters . . . . .	198
9.1.1	ASCII and Unicode . . . . .	198
9.1.2	The class Character . . . . .	201
9.2	Strings . . . . .	201
9.2.1	Constructing Strings . . . . .	202
9.2.2	String Accessors . . . . .	207

9.2.3	String Comparisons, Tests, and Searches . . . . .	209
9.2.4	String Hashing . . . . .	210
9.3	StringBuilders . . . . .	210
9.4	Readers, Writers, Streams, and Files . . . . .	211
9.4.1	Input . . . . .	215
9.4.2	Readers and InputStreams . . . . .	217
9.4.3	Output . . . . .	218
9.4.4	PrintStreams and PrintWriters . . . . .	218
9.5	Regular Expressions and the Pattern Class . . . . .	219
9.6	Scanner . . . . .	223
<b>10</b>	<b>Generic Programming</b>	<b>227</b>
10.1	Simple Type Parameters . . . . .	228
10.2	Type Parameters on Methods . . . . .	230
10.3	Restricting Type Parameters . . . . .	231
10.4	Wildcards . . . . .	232
10.5	Generic Programming and Primitive Types . . . . .	232
10.6	Formalities . . . . .	234
10.7	Caveats . . . . .	236
<b>11</b>	<b>Multiple Threads of Control</b>	<b>239</b>
11.1	Creating and Starting Threads . . . . .	242
11.2	A question of terminology . . . . .	243
11.3	Synchronization . . . . .	245
11.3.1	Mutual exclusion . . . . .	246
11.3.2	Wait and notify . . . . .	248
11.3.3	Volatile storage . . . . .	250
11.4	Interrupts . . . . .	251
	<b>Index</b>	<b>255</b>





# Chapter 1

## Java Overview

Different people have different ways of learning programming languages. Your author likes to read reference manuals (believe it or not)—at least if they are reasonably complete—on the grounds that this is the most efficient way to absorb a language quickly. Unfortunately, it is an approach that only works when one knows what to expect from a programming language, and has the necessary mental cubbyholes already constructed and ready for filing away specific details. Less grizzled programmers usually benefit from some kind of tutorial introduction, in which we look at examples of programs and program fragments. That’s the purpose of this chapter. After reading it, the succeeding chapters of this book ought to be somewhat more digestible.

### 1.1 Basic Program Structure

Let’s start with the traditional simplest program:

```
/* Sample Program #1 */
public class Hello {
    public static void main (String[] arguments) {
        System.out.println ("Hello, world!"); // Message + newline
    }
}
```

This example illustrates a number of things about Java:

**Java programs are collections of definitions.** In this case, we have definitions of a *class* named `Hello`, and a *function* (or *method*) named `main`.

**Definitions are grouped into classes.** Java programs may contain definitions of functions, variables, and a few other things. Every such definition is contained in (*is a member of*) some class.

**Dots indicate some kind of containment.** In the example above, `System` is a class and `out` is a variable defined in that class. We can think of that variable,

in turn, as referring to an object that has parts (*members*), one of which is the `println` method. So in general, you can read  $X.Y$  as “the  $Y$  that is (or whose definition is) contained in  $X$ .” We sometimes call ‘.’ the *selection operator*.

**The syntax “method-name(expressions)” denotes a function call.** (Also called *method call* or *method (or function) invocation*.) The program first evaluates the expressions, which are called *actual parameters*. In our example, there is only one (more would be separated by commas), and its value is a *string* of characters. Next, the value(s) of any actual parameters are handed off to (*passed*) to the indicated method, which then does whatever it does (unsurprisingly, `System.out.println` prints the string that you pass it, followed by an end-of-line.)

**Executing a complete program means calling a method named `main`.** (Not just any method called `main`, but we’ll get to that detail later). Appropriately enough, the method called is often referred to as the *main program*. You won’t always have to write a `main` method:

- Sometimes, your task is to write a *component* or a *package* of components for use in one or more other programs. These will consist of a set of classes, not necessarily containing a `main` method.
- Sometimes, you are writing something that is intended to be dropped into an existing *framework* of supporting components. For example, when you write a Java *applet* (a program intended to be run by a Web browser when one clicks on the appropriate thing) the browser will supply a standard `main`, and your applet provides one of the methods it calls.

But for *standalone* programs (also called *application* programs), `main` is where things start.

**Comments are surrounded by `/* */` or by `//` and the end of the line.** Only humans read comments; they have no effect on the program.

**A note on printing.** It isn’t theoretically central to the topic of programming, but the printing or other display of results is obviously rather important. Many introductory Java texts get rather hung up on the use of various GUI (Graphical User Interface) tools for displaying and inputting text, commands, or other communications. I’d rather keep it simple, so we’ll start with straightforward interaction with a textual display and keyboard. For output purposes, we use `System.out.println` as you’ve seen, which comes with a couple of useful variants. Here is an equivalent to the `main` method above:

```
public static void main (String[] args) {
    System.out.print ("Hello,");
    System.out.print (" world!");
    System.out.println ();
}
```

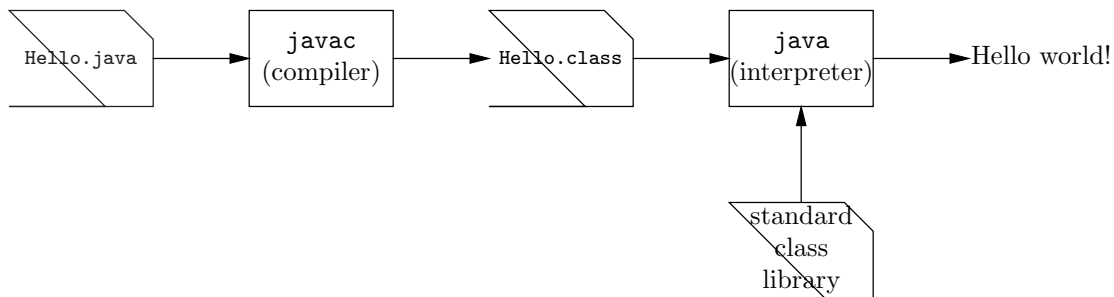


Figure 1.1: From program to execution. We first process our sample program with the `javac` compiler to create a class file, which we can then execute with an interpreter (`java`) to finally print a message. In this tiny program, pre-supplied programs from the standard class library actually do most of the work.

By comparing the two versions, you can probably deduce that the difference between `print` and `println` is that `println` ends the line immediately after printing, whereas `print` does not. You'll use `print` to break the printing of a complex line of output into pieces.

## 1.2 Compilation and Execution

Precise details on how to prepare and run this program differ from one implementation of Java to another. Here is a typical sequence.

- Put the program in one or more files. It generally makes things easier for the Java system if you put each class in a file named after the class—for our example, `Hello.java`. One can put several classes into a file or name the file arbitrarily, but the Java system will then sometimes be unable to find what it needs.
- *Compile* the file with a Java *compiler*. A compiler is a translation program that converts a *source program* (our little sample program is a source program) into some form that is more easily executed. Java compilers typically translate each class into a *class file*. For example, if we use Sun Microsystems's software, the command line

```
javac Hello.java
```

will translate class `Hello` into the file `Hello.class`.

- *Execute* the program by invoking a Java *interpreter*. An interpreter is a program that executes programs. In the case of Sun's software, the command

```
java Hello
```

runs our program, producing the output line

```
Hello, world!
```

The argument to the `java` program is the name of the class containing the desired `main` method. The interpreter finds out what it is supposed to do by reading the file `Hello.class`, together with a library of other class files that implement standard, pre-defined parts of Java (like the `System` class mentioned above).

There are numerous variations on this process. There are systems in which the compiler, rather than producing a `.class` file for input to an interpreter, produce a machine-language or *executable* file that the hardware of your machine can execute directly. This arrangement is typical for languages such as C and C++.

As another variation, the interpreter in many Java implementations produces machine code from `.class` files “just in time” for it to be executed when needed—hence the common term *JIT compiler*.

Typical *interactive development environments* (IDEs), such as Eclipse, tend to hide what’s really happening from you. You just write your program and push some button labeled “Run”, and the IDE compiles whatever is needed and executes it.

## 1.3 Simple Values and Expressions

You can think of a program as a set of instructions for creating *values* and moving them around, eventually sending them to something that does something visible with them, like displaying them, or moving some physical control. Values may either be *primitive*—in Java, these are various kinds of numbers, characters (as from an alphabet), and booleans (true/false)—or they may be *structured* into compound values. A programming language construct that produces a value is called an *expression*.

### 1.3.1 Writing Numbers

Something that denotes a number is known as a *numeral*, or, especially in programming languages, as a numeric *literal*. Java has both *integral* and *floating-point* literals (the difference being that the latter can have fractional parts). Here are some examples, which are mostly self-explanatory:

```
class Numerals {
    public static void main (String[] args) {
        // A. All print 42
        System.out.println (42);      // Base 10
        System.out.println (052);     // Base 8 (starts with 0)
        System.out.println (0x2a);    // Base 16 (a=10, b=11, ..., f=15)
        // B. All print 31.45
        System.out.println (31.45);
        System.out.println (3.145e1); // en or En means  $\times 10^n$ 
        System.out.println (3145e-2);
        // C. Prints 12345678901
        System.out.println (12345678901L);
    }
}
```

```
}

```

Technically, what we think of as negative numbers, such as `-3`, are actually positive literals operated on by the negation operator (`-`).

Example C indirectly illustrates a common feature of “production” programming languages such as C, C++, Java, Fortran, Pascal, Basic, and so on. For reasons having to do with fast and efficient execution, the “integers” in these programming languages comprise a tiny, finite portion of the mathematical integers. Ordinary integers in Java, for example, are all in the range  $-2^{31}$  to  $2^{31}-1$ , or about  $\pm 2$  billion. To get somewhat larger integers, you must use a distinct *type* of value, known as a `long`, whose literals have a trailing ‘L’ on them. These give you a range of  $-2^{63}$  to  $2^{63}-1$ . To go even further requires one of the built-in library classes (appropriately called `java.math.BigInteger`), for which there aren’t any built-in literals.

### 1.3.2 Arithmetic

Java expressions include the familiar algebraic notation, which you can play with easily enough by putting examples into a `main` function:

```
class Arith {
    public static void main (String[] args) {
        // A. Prints 3
        System.out.println ((3 + 7 + 10) * (1000 - 8)
                             / 992 - 17);
        // B. Prints 2.7166666666666663
        System.out.println (2.0 + 1.0/2 + 1.0/6 + 1.0/24 + 1.0/120);
        // C. Prints 2
        System.out.println (2 + 1/2 + 1/6 + 1/24 + 1/120);
        // D. Prints -3
        System.out.println (1 - 2 - 2);
        // E. Prints 17
        System.out.println (117 % 20);
        // F. Prints Infinity
        System.out.println (1.0 / 0.0);
        // G. Prints -2147483648
        System.out.println (2147483647 + 1);
        // H. Prints NaN
        System.out.println (0.0 / 0.0);
        // I. Halts the program with an exception
        System.out.println (1/0);
    }
}
```

As you can see, these examples look pretty much like what you might write by hand, and illustrate a few more points:

- ‘`*`’ denotes multiplication and ‘`%`’ denotes remainder.

- **Operators have precedences.** Example B illustrates *grouping*: the subexpression  $2.0 + 1.0/2$  is interpreted as  $2.0 + (1.0/2)$ , not  $(2.0 + 1.0)/2$ , because the division operator *groups more tightly* (or *has higher precedence* than) the addition operator. Example A uses parentheses to control grouping.
- **Operators associate.** Example D illustrates grouping when the operators have the same precedence. It is interpreted as  $(1 - 2) - 2$ , not  $1 - (2 - 2)$ , because subtraction (like most operators) *associates left* (or *groups left*) with operators of the same precedence.
- **Integer quantities and floating-point quantities are distinct.** In mathematics, the quantity written ‘1’ can be interpreted as either an integer or a real number, and the expression  $1/2$  always means the same as 0.5. In Java (as in most programming languages), a number written without a decimal point (an *integral quantity*) is of a different type from one written with a decimal point (a *floating-point quantity*), and the rules of arithmetic differ markedly. Division of two integers throws away the fractional part of the answer to yield an integer, whereas division of floating-point numbers behaves more like regular mathematics. Thus  $1/2$  yields 0, while  $1.0/2.0$  yields 0.5. When the two types are mixed, Java first converts the integer quantities to floating point.
- **Floating-point arithmetic approximates real arithmetic.** If you work out the mathematical value you’d normally expect from example B, you’ll see it differs slightly from the one printed, which appears to be off by about  $3.7 \times 10^{-16}$ . Floating-point arithmetic is a compromise between computational speed and mathematical truth. The results of each operation are computed, correctly rounded, to a certain number of binary digits (in this case, 52).
- **Arithmetic has limited range.** Example G illustrates that when an integer value becomes too large, the result “wraps around” to the smallest number. There are special values (which print as `Infinity` and `-Infinity`) to represent numbers that are too large in floating point, which are also used when non-zero floating-point numbers are divided by 0 (Example F).
- **Arithmetic nonsense has predictable effects.** Example H shows that a floating-point operation with an undefined value (there’s no sensible definition one can give to  $0.0/0.0$ ) yields a strange special value called NaN (for “Not A Number”). The analogous integer operation causes an error (what Java calls an *exception*), since for various reasons, there is no available integer to use as NaN.

### 1.3.3 Comparisons and Logical Operations

Comparison operators and logical operators produce the two boolean values `true` and `false`:

```
class Compare {
    public static void main (String[] args) {
```

```

    // A. All print true
    System.out.println (true);
    System.out.println (3 < 4);
    System.out.println (3 <= 4); System.out.println (3 <= 3);
    System.out.println (4 > 3);
    System.out.println (4 >= 3); System.out.println (3 >= 3);
    System.out.println (4 != 3); System.out.println (3 == 3);
    // B. All print false
    System.out.println (false);
    System.out.println (3 < 3); System.out.println (3 != 3);
    // C. All print true
    System.out.println (3 < 4 && 4 < 5);
    System.out.println (3 > 4 || 4 < 5);
    System.out.println (! (3 < 4 && 4 < 3));
    // D. All print true
    System.out.println (3 < 4 || 1/0 == 0);
    System.out.println (! (3 > 4 && 1/0 == 0));
}
}

```

Again, this is all pretty easy to figure out, once you see the transliterations Java uses for common notations:

$\leq \implies <=$	$\geq \implies >=$
$= \implies ==$	$\neq \implies !=$
and $\implies \&\&$	or $\implies   $
not $\implies !$	

All of the operators demonstrated have precedences below those of arithmetic operators, with the logical operators having the lowest precedences, *except* that logical not (!) has a precedence equal to that of the negation operator (-3). If that strikes you as potentially confusing, welcome to the club. Java has a large collection of operators at 15 different levels of precedence, which quickly become difficult to remember. As a stylistic matter, therefore, feel free to use parentheses even when operator-precedence rules make them redundant, if that makes your program clearer.

Example D illustrates that `&&` and `||` are “short-circuit” operators. Even though the second operand in both these cases would cause an exception, no such thing happens because the `||` only evaluates its right operand if needed. Upon evaluating its left operand, and finding it true, it skips evaluation of the right operand, while the `&&` doesn’t bother to evaluate its right operand if the left evaluates to false. This is an important property of these two operators; the first operand acts as a kind of *guard* on the second. All the other arithmetic and comparison operators, as well as method calls, always evaluate all their operands.

### 1.3.4 Strings

In programming-language jargon, a *string* is a sequence of *characters*. We write strings in double quotes, using a backslash where needed to indicate special characters, including double quotes and backslashes:

```
System.out.println ("Simple string.");
System.out.println ("");    // An empty string
System.out.println ("Say \"Hello.\""); // \" is double quote
System.out.println ("Name:\tJohn");   // \t is tab
System.out.println ("\\FOO\\BAR");
System.out.println ("One thing\nThe other thing");
```

These statements print (on Unix systems):

```
Simple string.

Say "Hello."
Name:   John
\\FOO\\BAR
One thing
The other thing
```

The newline character (notated `\n`) denotes the end-of-line on Unix systems. This is not, unfortunately, universal, which is why I've used `println` rather than `\n` to put things on separate lines (`println` “does the right thing” at the end of a line for whatever system you use).

Characters, the components of strings, have their own literals, which are enclosed in single quotes, with the same back-slashing notation:

```
'A'    '\t'    '\\ '    '\"'    '\n'
```

The `+` operator works on strings, but in that case means “concatenate”, as in

```
System.out.println ("The value of" +
                    " 17+25 is " + (17+25) + ".");
```

which prints

```
The value of 17+25 is 42.
```

You might well protest at this point that `(17+25)` yields an integer, not a string. This is correct; conveniently, though, the `+` operator is smart enough that when one of its operands is a string, and the other isn't, it converts the other operand to a string. In fact, underneath the hood, the `println` method ultimately prints strings, and the variations that work on other types first perform this same conversion.

There are quite a few other operations on strings, of which some of the more important are these:



<code>("19-character string").length ()</code>	$\Rightarrow$ 19	length of string
<code>("abcd").charAt (3)</code>	$\Rightarrow$ 'd'	get a character (number from 0)
<code>("abcd").equals ("abcd")</code>	$\Rightarrow$ true	compare contents
<code>("Hello, world!").substring (7, 12)</code>	$\Rightarrow$ "world"	substring between start and end positions
<code>("Hello, world!").substring (7)</code>	$\Rightarrow$ "world!"	substring starting at given position

The `.equals` method here is interesting. You might think that `==` would work, but for reasons we'll see later, that is not a reliable way to compare strings.

### 1.3.5 Static Methods: Abstracting Computation

Method (function) definition is the most basic *abstraction mechanism* in a typical programming language. It enables you to “define a new verb” for later use in your program, making it unnecessary to write, remember, or even know the details of how to perform some action.

In Java, there are *static* methods (also called *class methods*) and *non-static* (or *instance*) methods. We'll look at instance methods later. Static methods correspond to plain, ordinary functions, subprograms, or procedures in other programming languages (such as C, Pascal, Fortran, Scheme, or Basic). We've already seen examples of static method definition (of the `main` method).

The basic syntax is relatively easy:

```
A static Tr N (T1 N1, T2 N2, ...) {
    B
}
```

where

- $T_r, T_1$ , etc., denote *types* of values.  $T_r$  is the type of value returned (produced, yielded) by the method, and the other  $T_i$  are the *formal-parameter types*, the types of values that may be passed to the method. In Java, unlike dynamically typed languages such as Scheme or the scripting languages Python and Perl, one must be specific about the types of most quantities. You can indicate that a method does stuff (printing, for example), but does not return a value, by using the special “type” **void** for  $T_r$ .
- $A$  is one of the keywords **public**, **private**, **protected**, or it is simply missing (the default). It indicates the *access* of the method—what other parts of the program may call it. The **public** modifier, as you might guess, means that the method may be called from anywhere. The **private** modifier restricts access to the same class. (See §1.14)
- $N$  is the name by which the method is known. A method definition always occurs inside a class definition. The full name of a static method  $N$  defined in

a class  $C$  is  $C.N$  (dot means “that is defined in”). However, the simple name  $N$  works within the same class.

- $N_1$ , etc., are the *formal parameters*. These are names by which the body of the method ( $B$ ) can refer to the values passed to it.
- $\{ B \}$  is the *body* of the method.  $B$  is a sequence of statements that perform whatever computation the method is supposed to perform ( $B$  may be empty if the method does nothing).
- The *header* of the method definition—everything except the body—defines its *signature*, which consists of its name, return type, and formal-parameter types.

So far, the only statements we’ve seen are method calls (to `println`). We’ll see more later, but there’s one particularly important ones for methods that return values: the statement

```
return E;
```

means “end this call to the method, causing it to yield the value  $E$  (an expression).” Here are two examples:

```
/** The square of X. */
public static double square (double x) {
    return x*x;
}

/** The distance between points (X0, Y0) and (X1, Y1). */
public static double dist (double x0, double y0, double x1, double y1) {
    return Math.sqrt (square (x1 - x0) + square (y1 - y0));
    // NOTE: Math.sqrt is a standard square-root function */
}
```

These two examples also illustrate some *documentation conventions* that I’ll be using throughout. Comments beginning with ‘`/**`’ are, by convention, called *documentation comments*. They are to be used before definitions of things to describe what those things do or are for. In the case of methods, I refer to the parameter names by writing them in upper case (the idea being that this is a way to set them off from the rest of the text of the comment in a way that does not depend on fancy fonts, colors, or the like). At least one tool (called `javadoc`) recognizes these comments and can process them into on-line documentation.

## 1.4 Conditional Execution

To do anything very interesting, programs eventually have to make decisions—to perform different calculations depending on the data. There are several constructs to represent this in Java. Here, we’ll consider the simplest.

### 1.4.1 If statements

An *if-statement* has one of the forms

```
if (condition) then-part
// or
if (condition) then-part else else-part
```

(Normally, though, we lay these statements out on separate lines, as you'll see). The *condition* is a boolean expression. If it evaluates to **true**, the program executes the *then-part* (a statement), and otherwise the *else-part*, which simply defaults to the empty statement if absent.

We'll often need to have *then-part* or *else-part* contain more than one statement. The trick used for this purpose (and for other constructs that have a "statement" part that needs to be several statements) is to use a kind of statement called a *block*. A block consists of any number of statements (including zero) enclosed in curly braces { }. Unlike other statements you've seen, it does not end in a semicolon. For example:

```
if (3 > 4) {
    System.out.println ("3 > 4.");
    System.out.println ("Looks like we're in trouble.");
} else
    System.out.println ("OK");
```

### 1.4.2 Conditional Expressions

The expression  $C ? E_t : E_f$  yields either the value of  $E_t$  or  $E_f$ , depending on whether  $C$  evaluates to true or false. That is, '?' means roughly "then" and ':' means "otherwise", with an invisible "if" in front. This peculiar-looking ternary operator is not commonly used in C, C++, or Java, perhaps because it, well, looks peculiar. It is only your author's desire for balance that prompts its inclusion here. Here are some examples:

```
System.out.println (3 < 4 ? "Less" : "More"); // Prints Less
System.out.println (4 < 3 ? "Less" : "More"); // Prints More
// The next two mean the same, and print 3.1
System.out.println (1 < 0 ? 1.1 : 1 > 2 ? 2.1 : 3.1);
System.out.println (1 < 0 ? 1.1 : (1 > 2 ? 2.1 : 3.1));
```

The expressions after '?' and ':' must be the same type of thing; `3 < 4 ? 1 : "Hello"` is erroneous.

### 1.4.3 Case analysis and the Switch Statement

From time to time, you'll find yourself in a situation where you need to break some subproblem down into several cases. This is something to avoid if possible, but not all problems lend themselves to simple, unconditional formulae. You can always get by with multiple uses of the **if** statement, like this mathy example:

```

if (x < -10.0)
    y = 0.0;
else if (x < 0.0)
    y = 1.0 / f(-x);
else if (x < 10.0)
    y = f(x);
else
    y = Double.POSITIVE_INFINITY;

```

This compound **if** statement is so common that we customarily format as you see above, instead of indenting the entire **else** part of the first **if**, etc., like this:

```

// Example of how NOT to format your program
if (x < -10.0)
    y = 0.0;
else
    if (x < 0.0)
        y = 1.0 / f(-x);
    else
        if (x < 10.0)
            y = f(x);
        else
            y = Double.POSITIVE_INFINITY;

```

Sometimes, you'll encounter programs where first, there are *lots* of cases, and second, each test is of the form

```
if (E = some integral constant)
```

where *E* is the same each time. For this purpose, there is a special construct, introduced with the keyword **switch**. For example, consider this fragment:

```

if (action == PICK_UP)
    acquireObject ();
else if (action == DROP)
    releaseObject ();
else if (action == ENTER || action == EXIT)
    changeRooms (action);
else if (action == QUIT) {
    cleanUp ();
    return;
} else if (action == CLIMB)
    ...
else
    ERROR ();

```

Here, **action** is some integral variable (this might include character constants as well, but not strings), and the upper-case names are all defined in a class somewhere as *symbolic constants* standing for integers. The old-fashioned way to do this is

```

static final int  // 'final' means 'constant'
    QUIT = 0,
    PICK_UP = 1,
    DROP = 2,
    ENTER = 3,
    EXIT = 4,
    CLIMB = 5,
    etc.

```

As a matter of style, whenever you represent some set of possible values with arbitrary integers like this, always introduce meaningful symbols like these—avoid sprinkling your program with “mystery constants.” Recent versions of Java, however, have introduced *enumerated types* for this purpose. We’ll get to them in §5.6.

An arguably clearer way to write the **if** statement above is with **switch**, as follows:

```

switch (action) {
case PICK_UP:
    acquireObject ();
    break;
case DROP:
    releaseObject ();
    break;
case ENTER: case EXIT:
    changeRooms (action);
    break;
case QUIT:
    cleanUp ();
    return;
case CLIMB:
    ...
default:
    ERROR ();
}

```

Here, the **break** statements mean “leave this **switch** statement.”

## 1.5 Arrays I: The Command Line

Before finally turning to an example, we’ll need one other little piece of syntax to allow simple communication from the outside world to our program. I said earlier that the main program looks like this:

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

This notation means that the formal parameter, **args**, is an *array* of strings<sup>1</sup>. For our immediate purposes, this means that it references a sequence of string

---

<sup>1</sup>The name **args** is arbitrary, and you are free to choose another.

values, called `args[0]`, `args[1]`, etc., up to `args[args.length - 1]`. That is, `args.length` is the number of strings in `args`. When you start your program with

```
java MainClassName foo bar baz
```

the method `main` is called with "foo" as `args[0]`, "bar" as `args[1]`, and "baz" as `args[2]`.

You have to be careful to remember that the arguments to `main` are strings. The following main program won't work:

```
class Remainder {
    public static void main (String[] operands) {
        System.out.println (operands[0] % operands[1]);
    }
}
```

You might think that you could run this with

```
java Remainder 1003 12
```

and have it print 7. However, until told otherwise, the Java system will not assume that a string is to be treated as a number just because all its characters happen to be digits. The operation of converting a string of characters into a number requires specific action in Java:

```
class Remainder {
    public static void main (String[] operands) {
        System.out.println (Integer.parseInt (operands[0])
                           % Integer.parseInt (operands[1]));
    }
}
```

That is, the `Integer.parseInt` method converts a string of decimal digits to an integer. (There's a similar method `Double.parseDouble` for floating-point numbers).

## 1.6 Example: Finding Primes

You've now seen a great deal of the *expression language* of Java, enough in fact to write quite elaborate programs in the *functional* style. Let's start with a simple program to find prime numbers. A prime number is an integer larger than 1 whose only positive divisors are itself and 1<sup>2</sup>. Suppose that we want a program that, when given a number, will print all primes less than or equal to that number, perhaps in groups of 10 like this:

```
You type:  java primes 101
It types:   2 3 5 7 11 13 17 19 23 29
            31 37 41 43 47 53 59 61 67 71
            73 79 83 89 97 101
```

---

<sup>2</sup>Depending on whom you read, there can be negative primes as well, but for us, this would be an unnecessary complication, so we stick to positive primes.

### 1.6.1 Starting from the top

There are many ways to attack a simple problem like this. We may proceed *top-down*, starting with a sort of overall outline or skeleton, which we subsequently fill in, or *bottom-up*, building pieces we know we'll need and assembling them into a whole. Finally, we might use a little of both, as we'll do here. For example, the description above suggests the following program, with comments in place of stuff we haven't written yet:

```
class primes {
    /** Print all primes up to ARGS[0] (interpreted as an
     * integer), 10 to a line. */
    public static void main (String[] args) {
        printPrimes (Integer.parseInt (args[0]));
    }

    /** Print all primes up to and including LIMIT, 10 to
     * a line. */
    private static void printPrimes (int limit) {
        // do the work
    }
}
```

This is a simple example of the classic “put off until tomorrow,” top-down school of programming, in which we simplify our task in stages by assuming the existence of methods that do well-defined pieces of it (writing down their definitions, not including their bodies, but including comments that will remind us what they're supposed to do). Then we handle each incomplete piece (sometimes called an *obligation*) by the same process until there are no more incomplete pieces

### 1.6.2 Starting from the bottom

Our problem involves prime numbers, so it might well occur to you that it would be useful to have a method that tests to see if a number is prime. Let's proceed, bottom-up now, to define one, hoping to tie it in later to program we've started building from the top.

```
/** True iff X is a prime number. */
private static boolean isPrime (int x) {
    return /*( X is prime )*/;
}
```

I've introduced a piece of pseudo-Java here to help in developing our program: the notation

```
/*( description of a value )*/
```

is intended to denote an obligation to produce some expression (as yet undetermined) that yields a value fitting the given description. The description “ $X$  is prime” is a true/false sentence, so in this example, `isPrime` yields true iff  $X$  is prime<sup>3</sup>:

One general technique is to work backward from possible answers to the conditions under which those answers work. For example, we know immediately that a number is *not* prime—that we should return **false**—if it is  $\leq 1$ , so we can replace the dummy comment as follows:

```
if (x <= 1)
    return false;
else
    return /*( True iff X is prime, given that X>1 )*/;
```

I’ll call this the *method of guarded commands*: the “guards” here are the conditions in the **if** statements, and the “commands” are the statements that get executed when a condition is true.

For larger numbers, the obvious brute-force approach is to try dividing  $x$  by all positive numbers  $\geq 2$  and  $< x$ . If there are none, then  $x$  must be prime, and if any of them divides  $x$ , then  $x$  must be composite (non-prime). Let’s put this task off until tomorrow, giving us our finished `isPrime` method:

```
private static boolean isPrime (int x) {
    if (x <= 1)
        return false;
    else
        return ! isDivisible (x, 2);
}

/** True iff X is divisible by any positive number >=K and < X,
 *  given K > 1. */
private static boolean isDivisible (int x, int k) {
    /*{ return true iff x is divisible by some integer j, k ≤ j <
    x; }*/
}
```

Here, I am using another piece of pseudo-Java: the comment

```
/*{ description of an action }*/
```

is intended to denote an obligation to produce one or more statements that perform the described action.

By the way, we saw conditional expressions in §1.4. They allow an alternative implementation of `isPrime`:

```
private static boolean isPrime (int x) {
    return (x <= 1) ? false : ! isDivisible (x, 2);
}
```

---

<sup>3</sup>The notation *iff* used here and elsewhere is mathematician’s shorthand for “if and only if.”



To tackle this new obligation, we can again use guarded commands. For one thing, we know that `isDivisible` should return **false**, according to its comment, if  $k \geq x$ , since in that case, there aren't any numbers  $\geq k$  and  $< x$ :

```

    if (k >= x)
        return false;
    else
        /*{ return true iff x is divisible by some integer j,
           where  $k \leq j < x$ , assuming that  $k < x$ ; }*/

```

Now, if we assume that  $k < x$ , then (always following the documentation comment) we know that one case where we should return **true** is where  $k$  divides  $x$ :

```

    if (k >= x)
        return false;
    else if (x % k == 0)
        return true;
    else
        /*{ return true iff x is divisible by some integer j,  $k \leq j < x$ ,
           assuming  $k < x$ , and x is not divisible by k; }*/

```

Here, I've used the remainder operator to test for divisibility.

Well, if  $k$  itself does not divide  $x$ , then for  $x$  to be divisible by some number  $\geq k$  and  $< x$ ,  $x$  must be divisible by a number  $\geq k + 1$  and  $< x$ . How do we test for that? By a curious coincidence, we just happen to have a method, `isDivisible`, whose comment says that if given two parameters,  $x$  and  $k$ , it returns "True iff  $x$  is divisible by any positive number  $\geq k$  and  $< x$ , given  $k > 1$ ." So our final answer is

```

/** True iff X is divisible by any positive number  $\geq K$  and  $< X$ ,
 *  given  $K > 1$ . */
private static boolean isDivisible (int x, int k) {
    if (k >= x)
        return false;
    else if (x % k == 0)
        return true;
    else
        return isDivisible (x, k+1);
}

```

Now you might object that there's something circular about this development: to finish the definition of `isDivisible`, we used `isDivisible`. But in fact, the reasoning used is impeccable: *assuming* that `isDivisible` does what its comment says, the body we've developed can only return a correct result. The only hole through which an error could crawl is the question of whether `isDivisible` ever gets around to returning a result at all, or instead keeps calling itself, forever putting off until tomorrow the production of a result.

However, you can easily convince yourself there is no problem here. Each time `isDivisible` gets called, it is called on a problem that is “smaller” in some sense. Specifically, the difference  $x - k$  gets strictly smaller with each call, and (because of the guard  $k \geq x$ ) must stop getting smaller once  $x - k \leq 0$ . (Formally, we use the term *variant* for a quantity like  $x - k$ , which shrinks (or grows) with every call but cannot pass some fixed, finite limit.) We have established that `isDivisible` is not circular, but is a proper *recursive* definition.

### 1.6.3 Meeting in the middle

We are left with one final obligation, the `printPrimes` procedure:

```
/** Print all primes up to and including LIMIT, 10 to a line. */
private static void printPrimes (int limit) {
    /*{ do the work }*/
}
```

Let’s make the reasonable assumption that we’ll be printing prime numbers one at a time by looking at each number up to the limit, and figuring out whether to print it. What do we need to know at any given time in order to figure out what to do next? First, clearly, we need to know how far we’ve gotten so far—what the next number to be considered is. Second, we need to know what the limit is, so as to know when to stop. Finally, we need to know how many primes we’ve already printed, so that we can tell whether to start a new line. This all suggests the following refinement:

```
private static void printPrimes (int limit) {
    printPrimes (2, limit, 0);
    System.out.println ();
}

/** Print all primes from L to U, inclusive, 10 to a line, given
 *  that there are initially M primes printed on the current
 *  line. */
private static void printPrimes (int L, int U, int M) {
    ...
}
```

This illustrates a small point I hadn’t previously mentioned: it’s perfectly legal to have several methods with the same name, as long as the number and types of the arguments distinguish which you intend to call. We say that the name `printPrimes` here is *overloaded*.

To implement this new method, we once again employ guarded commands. We can easily distinguish four situations:

- A.  $L > U$ , in which case there’s nothing to do;
- B.  $L \leq U$  and  $L$  is not prime, in which case we consider the next possible prime;

- C.  $L \leq U$  and  $L$  is prime and  $M$  is less than 10, in which case we print  $L$  and go on to the next prime); and
- D.  $L \leq U$  and  $L$  is prime and  $M$  is divisible by 10, which is like the previous case, but first we go to the next line of output and adjust  $M$ ).

A completely straightforward transliteration of all this is:

```

if (L > U)                                // (A)
    ;
if (L <= U && ! isPrime (L))              // (B)
    printPrimes (L+1, U, M);
if (L <= U && isPrime (L) && M != 10) {    // (C)
    System.out.print (L + " ");
    printPrimes (L+1, U, M+1);
}
if (L <= U && isPrime (L) && M == 10) {    // (D)
    System.out.println ();
    System.out.print (L + " ");
    printPrimes (L+1, U, 1);
}

```

Well, this is a little wordy, isn't it? First of all, the conditions in the **if** statements are mutually exclusive, so we can write this sequence more economically as:

```

if (L > U)                                // (A)
    ;
else if (! isPrime (L))                   // (B)
    printPrimes (L+1, U, M);
else if (M != 10) {                       // (C)
    System.out.print (L + " ");
    printPrimes (L+1, U, M+1);
} else {                                  // (D)
    System.out.println ();
    System.out.print (L + " ");
    printPrimes (L+1, U, 1);
}

```

Step (A) looks kind of odd. We often write things like this as

```

if (L > U)
    return;

```

to make it clearer that we're done at this point, or we can *factor* the condition like this:

```

if (L <= U) {

```

```

    if (! isPrime (L))                                // (B)
        printPrimes (L+1, U, M);
    else if (M != 10) {                                // (C)
        System.out.print (L + " ");
        printPrimes (L+1, U, M+1);
    } else {                                           // (D)
        System.out.println ();
        System.out.print (L + " ");
        printPrimes (L+1, U, 1);
    }
}

```

I use the term “factor” here in a sort of arithmetical sense. Just as we can rewrite  $ab + ac + ad$  as  $a(b + c + d)$ , so also we can “factor out” the implicit  $!(L \leq U)$  from all the other conditions and just drop the  $L > U$  case.

Finally, recognizing the  $M$  keeps track of the number of primes already printed on the current line and doing a little more factoring, we get this:

```

if (L <= U) {
    if (! isPrime (L))                                // (B)
        printPrimes (L+1, U, M);
    else {
        if (M == 10) {
            System.out.println ();
            M = 0;
        }
        System.out.print (L + " ");
        printPrimes (L+1, U, M+1);
    }
}

```

## 1.7 Example: Pig Latin

We’d like to write a program (never mind why) that translates into “Pig Latin.” There are numerous dialects of Pig Latin; we’ll use the simple rule that one translates into Pig Latin by moving any initial consonants to the end of the word, in their original order, and then appending “ay.” So,

*You type:*    `java pig all gaul is divided into three parts`

*It types:*    `allay aulgay isay ivededday intoay eethray artspay`

We’ll simplify our life further by agreeing that ‘y’ is a consonant.

### 1.7.1 Vowels

This time, let’s start from the bottom. Presumably, we’ll need to know whether a letter is a consonant. Here’s a brute-force way:

```
static boolean isVowel (char x) {
    return x == 'a' || x == 'e' || x == 'i' || x == 'o' || x == 'u';
}
```

(This method has *default access*; I haven't labeled it specifically **private** or **public**. Most of the time, in fact, this is fine, although it is often good discipline (that is, leads to cleaner programs) to keep things as private as possible.)

**Aside: Tedious programs.** When you find yourself writing a particularly tedious expression, it's often a hint to look around for some shortcut (you will probably spend longer looking for the shortcut than you would grinding through the tedium, but you are likely to learn something that you can use to relieve later tedium). For example, if you browse through the detailed documentation for the `String` type (see, for example, §9.2), you'll find the useful function `indexOf`, which returns the first point in a string where a given character appears, or `-1` if it doesn't appear at all. Then the `return` statement above is simply

```
return ("aeiou").indexOf (x) >= 0;
```

### 1.7.2 Translation

Now let's turn to the problem of actually re-arranging a string. We've already seen all the parts: `substring` will extract a piece of a string, and `+` will concatenate strings. So, if we just knew how many consonants there were at the beginning of a word, `w`, the translation would just be

```
w.substring (k) + w.substring (0, k) + "ay"
```

where `k` is the number of consonants at the beginning of `w`. This expression works for any value of `k` between 0 and the length of `w`, inclusive. But which value to choose? Again, we put that off until tomorrow:

```
/** Translation of W (a single word) into Pig Latin. */
static String toPig (String w) {
    return w.substring (consonants (w)) + w.substring (0, consonants (w)) + "ay";
}

/** The number of consonants at the beginning of W (a single word). */
static int consonants (String w) {
    /*{ return the number of consonants at start of W }*/;
}
```

### 1.7.3 Counting Consonants

Assume now that tomorrow has arrived, let's consider how to implement `consonants`, again using a guarded command approach. Look for easy (or "base") cases. Clearly, we return 0 if `w` is empty:

```

if (w.length () == 0)
    return 0;
else
    /*{ return the number of consonants at start of W,
       assuming W is not empty }*/;

```

Likewise, in the remaining case, we should return 0 if `w` starts with a vowel:

```

if (w.length () == 0)
    return 0;
else if (isVowel (w.charAt (0)))
    return 0;
else
    /*{ return the number of consonants at start of W,
       assuming W starts with a consonant }*/;

```

Finally, a word that starts with a consonant clearly has one more consonant at the beginning than the “tail” of that word (the word with its first letter removed):

```

if (w.length () == 0)
    return 0;
else if (isVowel (w.charAt (0)))
    return 0;
else
    return 1 + consonants (w.substring (1));

```

and there’s nothing left to do. Optionally, we can condense things a bit, if desired:

```

if (w.length () == 0 || isVowel (w.charAt (0)))
    return 0;
else
    return 1 + consonants (w.substring (1));

```

Again, this program exemplifies a typical feature of the functional programming style: we’ve assembled a method from a collection of *facts* about the problem:

- An empty string has no consonants at the beginning.
- A string that starts with a vowel has no consonants at the beginning.
- A string that isn’t empty and doesn’t start with a vowel has one more consonant at the beginning than its tail does.

We could actually have written these independently and in any order and with redundant guards; for example:

```

if (w.length () > 0 && !isVowel (w.charAt (0)))
    return 1 + consonants (w.substring (1));
else if (w.length () > 0 && isVowel (w.charAt (0)))
    return 0;

```

```

else if (w.length () == 0)
    return 0;
// Should never get here.

```

(Java compilers, being none too bright, will complain that we might not execute a **return**, but we know the cases are exhaustive.) However, we usually take advantage of the cases we've already eliminated at each step to shorten the tests.

**Cutting expenses.** As it happens, the `substring` method in Java is considerably more expensive than `charAt`. So we might consider ways to use the latter exclusively. For example,

```

/** The number of consonants at the beginning of W (a single word). */
static int consonants (String w) {
    return consonants (w, 0);
}

/** The number of consonants at the beginning of the substring of W
 *  that starts at position K. */
static int consonants (String w, int k) {
    /*{ return the number of consonants immediately at and
       after position k start of w }*/;
}

```

For the body of this second `consonants` method, we can use the the same facts as before, expressed slightly differently:

```

if (w.length () <= k || isVowel (w.charAt (k)))
    return 0;
else
    return 1 + consonants (w, k+1);

```

This new `consonants` method is a *generalization* of the original; it can count consonants anywhere in a word, not just at the front. As a result, the resulting function is arguably smaller and cleaner than our first version. Generalization often has this effect (and just as often makes things more complex than they have to be—programming will always require judgment).

#### 1.7.4 To the top

Finally, we want to apply `toPig` to each of the command-line arguments and print the results. With what you've seen so far, an approach rather like the last version of `consonants` seems reasonable:

```

class pig {
    public static void main (String[] words) {
        translate (words, 0);
        System.out.println ();
    }
}

```

```

    }

    /** Print the translations of SENT[K], SENT[K+1], ..., all on one line */
    static void translate (String[] sent, int k) {
        ...
    }

```

In this case, `translate` doesn't have to do anything if `k` is already off the end of the array `sent` (for "sentence"). Otherwise, it has to print the translation of word `#k`, followed by the translations of the following words. So,

```

    if (k < sent.length) {
        System.out.print (toPig (sent[k]) + " ");
        translate (sent, k+1);
    }

```

which completes the program.

## 1.8 Variables and Assignment

Method signatures and documentation comments are abstractions of actions. When you write `translate` in the main program of §1.7, you should be thinking of "translation into Pig Latin," and not the details of this procedure, because *any* method that correctly translates a word into Pig Latin would serve the main program, not just the one you've happened to write. Likewise, the formal parameters of methods are abstractions of values. For example, `k` in the `translate` method of the last section refers to "the index of the first word to translate," rather than any particular value.

Formal parameters are a particular kind of *variable*. Although in some ways similar to mathematical variables, they aren't quite the same thing: whereas in mathematics, a variable denotes *values*, in computer languages they denote *containers of values*. The difference is that computer languages provide an *assignment operation*, which in Java is written `=` (a choice inherited from Fortran, and requiring the use of `==` for equality). The expression

```
x = 43
```

has 43 as its value, but also has the *side effect* of placing the value 43 in the container `x`. Until this value is replaced by another assignment, any use of `x` yields the value 43. An expression such as

```
x = x + 2;
```

first finds the value (contents) of `x`, adds 2, and then places the result back in `x`. This particular sort of statement is so common that there is a shorthand<sup>4</sup>:

---

<sup>4</sup>You'll find that most C, C++, and Java programmers also use another shorthand, `x++` or `++x`, to increment `x` by one. You won't see me use it, since I consider it a superfluous and ugly language misfeature.



```
x += 2
```

as well as `-=`, `*=`, etc.

This last example demonstrates something confusing about variables: depending on where it is used, `x` can either mean “the container named `x`” or “the value contained in the container named `x`”. It has the first meaning in what are called *left-hand side* contexts (like the left-hand side of an assignment operator) and the second in *right-hand side* or *value* contexts.

You can introduce new variables as temporary quantities in a method by using a *local-variable declaration*, with one of the following forms:

```
Type variable-name;
Type variable-name = initial-value;
```

For example,

```
int x;
String[] y;
double pi = 3.1415926535897932383;
double pi2 = pi * pi;
```

A declaration with initialization is just like a declaration of a variable followed by an assignment. With either of these forms, you can save some typing by re-using the *Type* part:

```
int x, y;
int a = 1, b = 2, c;
```

Be careful with this power to abbreviate; compactness does not necessarily lead to readability.

If you’re used to typical scripting languages (like Perl or Python), or to Lisp dialects (such as Scheme), you may find it odd to have to say what type of value a variable contains. However, this requirement is common to most “production” languages these days. The original purpose of such *static typing* was to make it easier for compilers to produce fast programs. Although some authors praise the “radical” and “liberating” alternative of having no restrictive variable types (*dynamic typing*), experience suggests that static typing provides both useful internal consistency checking and documentation that is of increasing value as programs start to become large.

With the program features we’ve seen so far, local variables and the assignment statement are theoretically unnecessary. When we look at iterative statements in the next section, you’ll see places where local variables and assignment become necessary. Still, they can provide shorthand to avoid repeating expressions, such as in this rewrite of a previous example

```
static String toPig (String w) {
    int numCon = consonants (w);
    return w.substring (numCon) + w.substring (0, numCon) + "ay";
}
```

or this example, involving an `if` statement:

<b>Version 1:</b>	<pre> if (x % k == 0)     System.out.println (x + " is divisible by " + k); else     System.out.println (x + " is not divisible by " + k); </pre>
<hr/>	
<b>Version 2:</b>	<pre> String sense; if (x % k == 0)     sense = " "; else     sense = " not "; System.out.println (x + " is" + sense + "divisible by " + k); </pre>

They may also be used for breaking up huge, unreadable expressions into bite-sized pieces:

<b>Version 1:</b>	<pre> tanh = u / (1.0 + u*u / (3.0 + u*u / (5.0 + u*u / 7.0))); </pre>
<hr/>	
<b>Version 2:</b>	<pre> double u2 = u*u, tanh; tanh = 7.0; tanh = 5.0 + u2/tanh; tanh = 3.0 + u2/tanh; tanh = 1.0 + u2/tanh; tanh = u / tanh; </pre>

This example also shows another use of a variable (`u2`) to represent a repeated expression.

## 1.9 Repetition

In the Scheme language, a *tail recursive* method (one in which the recursive call is the last operation in the method, like `printPrimes` in §1.6.3) is always implemented in such a way that the recursive calls can go on indefinitely (to any *depth*)—in Scheme, `printPrimes` would consume the same amount of computer memory for all values of its arguments. The typical imperative “production” languages in use today (including Java) do not put this requirement on their compilers, so that it is possible for the original Java version of `printPrimes` to fail for large values of `limit`. To a large extent, this difference is historical. Defining a function in the early algebraic languages (from which Java, C, and C++ descend) was a much less casual occurrence than in current practice and functions tended to be larger. Programmers relied on other methods to get repetitive execution. That tendency has survived, although the culture has changed so that programmers are more willing to introduce function definitions.

As a result, most commonly used languages include constructs for expressing repetition (or *iteration*). These come in two flavors: those where the (maximum)

number of repetitions is explicit (*definite iteration*) and those in which repetition terminates when some general condition is met and the number of repetitions is not directly expressed (*indefinite iteration*).

### 1.9.1 Indefinite iteration

Consider the problem of computing the gcd (greatest common divisor) of two non-negative integers—the largest integer that divides both. Recursively, this is easy using the method of guarded commands. First, we notice that if one of the numbers is 0, then (since every positive integer divides 0), the other number must be the gcd:

```
/** The greatest common divisor of A, and B, where A,B >= 0 */
static int gcd (int a, int b)
{
    if (b == 0)
        return a;
    else
        return /*( the gcd of a and b, given that b != 0 )*/;
}
```

Next, we use the fact that if  $a = r \bmod b$  for the remainder  $0 \leq r < b$ , then the gcd of  $b$  and  $r$  is the same as the gcd of  $a$  and  $b$ :

```
/** The greatest common divisor of A, and B, where A,B >= 0 */
static int gcd (int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

[Why does this terminate? What is the variant here?]

Were you to do this on a small slate (with limited space), you might end up re-using the same space for each new set of values for  $a$  and  $b$ . You would then be treating  $a$  and  $b$  as local variables and you would think of the process as something like “keep repeating this calculation, changing  $a$  and  $b$  until  $b = 0$ .” In Java, we might write this:

```
static int gcd (int a, int b) {
    while (b != 0) {
        int b1 = a % b;
        a = b; b = b1;
    } // Now b == 0
    return a;
}
```

This last program's relationship to the preceding recursive version might be a little clearer if we re-write the former like this:

```
static int gcd (int a, int b)
{
    if (b != 0)
        return gcd (b, a % b);

    return a;
}
```

A statement of the form

```
while (condition) statement
```

means “if *condition* evaluates to false, do nothing (“exit the loop”); otherwise execute *statement* (called the *loop body*) and then repeat the entire process.” Another way to say the same thing is that the **while** loop is equivalent to

```
if (condition) {
    statement
    while (condition) statement
}
```

This is a recursive definition, with the inner **while** loop functioning as the recursive “call.”

The **while** loop tests whether to terminate before each iteration. It is sometimes desirable to test *after* each iteration, so that the loop body is always executed at least once. For example, in a program that communicates with its users using a simple text-based interface, you'll often want to prompt for a response, read the response, and repeat the process until the response is legal. For this purpose, there is a **do-while** loop:

```
String response;
do {
    System.out.print ("Proceed? [y/n] ");
    response = readWord ();
} while (! (response.equals ("y") || response.equals ("n")));
```

(Assume that `readWord` reads the next whitespace-delimited word typed as input.) This loop corresponds to the following recursive function:

```
static String getResponse () {
    System.out.print ("Proceed? [y/n] ");
    String response = readWord ();
    if (response.equals ("y") || response.equals ("n"))
        return response;
    else
        return getResponse ();
}
```

Many loops fall into neither of these categories. Instead, one discovers whether to stop only in the middle of a computation. For this purpose, Java has a general-purpose exiting construct, **break**, which in its simplest form ends execution of the enclosing loop. For example, suppose that we want to prompt for and read a sequence of words from some input source and print their Pig Latin translations, one per line, until we get to a single period. We could write

```
String word;
System.out.print("> ");
word = readWord ();
while (! word.equals(".")) {
    System.out.println(word + " => " + toPig (word));
    System.out.print("> ");
    word = readWord ();
}
```

but the repetition of the statements for prompting and reading is a little annoying. With **break**, we can use a “loop and a half,” like this:

```
while (true) {
    System.out.print("> ");
    word = readWord ();
    if (word.equals("."))
        break;
    System.out.println(word + " => " + toPig (word));
}
```

The “**while (true)**” statement loops indefinitely, since its condition is, of course, never false. It is only stopped by the **break** statement in the middle of the loop body.

---

**Question:** What do these two loops do?

<pre>while (false) {     System.out.println ("Hello!"); }</pre>	<pre>do {     System.out.println ("Hello!") } while (false);</pre>
---	--

---

### 1.9.2 Definite Iteration

Consider the computation of  $N!$ , the factorial function, which for non-negative integers is defined by the recurrence

$$\begin{aligned} 0! &= 1 \\ (N+1)! &= (N+1) \cdot N! \end{aligned}$$

or to write it out explicitly<sup>5</sup>,

$$N! = 1 \cdot 2 \cdots N = \prod_{1 \leq k \leq N} k.$$

The recurrence suggests a recursive definition<sup>6</sup>:

```
/** N!, for N >= 0. */
static long factorial (long N) {
    if (N <= 0)
        return 1;
    else
        return N * factorial (N-1);
}
```

I’ve chosen to use type `long` because the factorial function grows quickly, and `long` is an integer type that accommodates values up to  $2^{63} - 1$  (slightly less than  $10^{19}$ ).

Now this last function is not tail-recursive; it multiplies by `N` after the recursive call. A simple trick makes it tail recursive—we compute a *running product* as an extra parameter:

```
static long factorial (long N) {
    return factorial (N, 1);
}

/** N! * P, assuming N >= 0. */
static long factorial (long N, long p) {
    if (N <= 0)
        return p;
    else return factorial (N-1, p*N);
}
```

This function is now amenable to being converted to a loop directly:

```
static long factorial (long N0) {
    long p, N;
    N = N0; p = 1;
    while (N > 0) {
        p *= N;      // Means p = p*N
        N -= 1;      // Means N = N-1
    }
    return p;
}
```

---

<sup>5</sup>The symbol  $\prod$  means “product of” just as  $\sum$  means “sum of.” When  $N$  is 0, so that you have an empty product, mathematical convention makes the value of that product 1, the multiplicative identity. By the same token, an empty sum is 0 by convention, the additive identity.

<sup>6</sup>This function yields 1 if given a negative argument. Since negative arguments are illegal (says the comment), the function may yield any result—or even fail to terminate—and still conform to the “letter of the specification.” Therefore, substituting  $\leq$  for  $=$  in the test is perfectly correct.

The call `factorial(N,1)` corresponds to initializing `p` to 1. This is a typical example of a *reduction loop* used to perform a certain binary operation on a sequence of values. The variable `p` serves as the *accumulator*.

This loop fits a very common pattern: one variable (here, `N`) is first initialized, and then incremented or decremented each time through a loop until it reaches some limit. The loop therefore has a predictable maximum number of iterations each time it is started. Numerous programming languages (FORTRAN, Algol, Pascal, and Ada, to name just a few) have special *definite iteration* constructs for just such combinations of actions. The Ada version, for example is

```
p := 1;
for N in reverse 1 .. NO loop
  p := p*N;
end loop;
```

Here, `N` is a *loop control variable*, which is modified only by the loop statement and may not be assigned to.

Instead of having a restricted definite iteration like other languages, Java (as in C and C++) has a more general form, in which we explicitly write the initialization, bounds test, and increment in one statement:

```
long p, N;
p = 1;
for (N = NO; N > 0; N -= 1)
  p *= N;
return p;
```

This **for** statement is little more than a compact shorthand for a **while** loop with some initial stuff at the beginning. In general,

$$\begin{array}{ccc} \text{for } (S_0; C; S_1) & \iff & \begin{array}{l} \{ S_0; \\ \text{while } (C) \{ \\ \quad B \\ \quad S_1 \\ \} \} \end{array} \\ B & & \end{array}$$

If you take this equivalence literally, you'll see that it's also legal to write

```
long p;
p = 1;
for (long N = NO; N > 0; N -= 1)
  p *= N;
```

declaring the loop-control variable in the loop itself.

In this case, since multiplication is commutative, we can also write

```
long p;
p = 1;
for (long N = 1; N <= NO; N += 1)
  p *= N;
```

and get the same effect. For whatever reason, incrementing loops like this tend to be more common than decrementing loops.

The three parts of a **for** statement all have defaults: the test is **true** if omitted, and the other two parts default to the null statement, which does nothing. Thus, you will find that many programmers write an infinite loop (**while** (**true**)...) as

```
for (;;) ...
```

### 1.9.3 Example: Iterative Prime-Finding

Let's look at what happens when we replace the recursive methods in §1.6 with iterative constructs. First, let's start with the original prime-testing methods from that section, re-arranged slightly to make the conversion clear:

```
private static boolean isPrime (int x) {
    if (x <= 1)
        return false;
    else
        return ! isDivisible (x, 2);
}

private static boolean isDivisible (int x, int k) {
    if (k < x) {
        if (x % k == 0)
            return true;
        else
            return isDivisible (x, k+1);
    }
    return false;
}
```

Using essentially the same techniques we used for factorial, we get the following iterative version for `isDivisible`:

```
private static boolean isDivisible (int x, int k) {
    while (k < x) {
        if (x % k == 0)
            return true;
        k += 1;
    }
    return false;
}
```

which we may also code as a **for** loop with a null initialization clause:

```
for (; k < x; k += 1)
    if (x % k == 0)
        return true;
return false;
```



Now we can actually get rid of the `isDivisible` method altogether by integrating it into `isPrime`:

```
private static boolean isPrime (int x) {
    if (x <= 1)
        return false;
    for (int k = 2; k < x; k += 1)
        if (x % k == 0)
            return false;
    return true;
}
```

Since `isPrime` evaluates `! isDivisible(x, 2)`, we had to be careful also to reverse the sense of some of the return values.

Similar manipulations allow us also to convert `printPrimes`:

```
/** Print all primes up to and including LIMIT, 10 to a line. */
private static void printPrimes (int limit) {
    int n;
    n = 0;
    for (int L = 2; L <= U; L += 1) {
        if (isPrime (L)) {
            if (n != 0 && n % 10 == 0)
                System.out.println ();
            System.out.print (L + " ");
            n += 1;
        }
    }
    System.out.println ();
}
```

(The full syntax of **for** actually allows us to make this even shorter by replacing the first three lines of the body with

```
for (int L = 2, n = 0; L <= U; L += 1) {
```

but I am not sure that this is any clearer.)

#### 1.9.4 Example: Counting

The consonant-counting method in the Pig Latin translator (§1.7.3) looked like this:

```
/** The number of consonants at the beginning of W (a single word). */
static int consonants (String w) {
    return consonants (w, 0);
}

static int consonants (String w, int k) {
```

```

    if (w.length () <= k || isVowel (w.charAt (k)))
        return 0;
    else
        return 1 + consonants (w, k+1);
}

```

This looks like a reduction loop, as introduced in §1.9.2, except that we stop prematurely at the first vowel. Indeed, we can use the same structure:

```

static int consonants (String w) {
    int n;
    n = 0;
    for (int k = 0; k < w.length () && ! isVowel (w.charAt (k)); k += 1)
        n += 1;
    return n;
}

```

## 1.10 Arrays II: Creation

You saw arrays in §1.5, where they were used as the argument to the main program. Array types are the most basic type used to hold sequences of items. If  $T$  is any type (so far, we’ve seen `int`, `long`, `char`, `double`, `boolean`, and `String`), then ‘ $T[]$ ’ denotes the type “array of  $T$ .” This definition is completely recursive, so that ‘ $T[][]$ ’ is “array of array of  $T$ .”

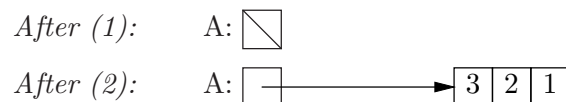
The array types are examples of what are called *reference types* in Java. This means that a declaration such as

```
int[] A;    // (1)
```

tells us that at any given time, **A** contains either a *reference* (or *pointer*) to an array of `ints`, or it contains something called the *null pointer*, which points at nothing (not even at an array containing 0 `ints`). For example, **A** declared above initially contains a null pointer, and after

```
A = new int[] { 3, 2, 1 };    // (2)
```

it contains a pointer to an array containing 3 `ints`: `A[0]==3`, `A[1]==2`, `A[2]==1`. You should keep pictures like this in your mind:



As illustrated in statement (2) above, one creates a new array for **A** to point to with the **new** operator. More commonly, you’ll simply want to create an array of a given size first, and fill in its contents later. So you’ll often see statements or declarations like these:

```
A = new int[100];
int[] B = new int[A.length];
```

which make A and B point at 100-element arrays, initially containing all 0's.

The syntax in (2) is convenient when using arrays to contain constant *tables* of values. For this purpose, Java (following C) allows a slightly condensed syntax when (and only when) you combine array declaration with creation. The following declaration, for example, is equivalent to (2):

```
int[] A = { 3, 2, 1 };
```

### 1.10.1 Example: Linear Interpolation

*Interpolation* is a procedure for approximating the value of a function at some point where its value is unknown from its (known) values at surrounding, known points. If the function is reasonably well-behaved (that is, continuous, and with a sufficiently small second derivative), one can *linearly interpolate*  $y = f(x)$ , given  $y_i = f(x_i)$  and  $y_{i+1} = f(x_{i+1})$ , with  $x_i \leq x \leq x_{i+1}$  and  $x_i \neq x_{i+1}$  as

$$y \approx y_i + (y_{i+1} - y_i) \cdot \frac{x - x_i}{x_{i+1} - x_i} \quad (1.1)$$

We'd typically have an ascending sequence of  $x_i$  and would choose the two that bracketed our desired  $x$ .

Suppose we'd like to write a program to carry out linear interpolation. Actually, with such a common, useful, and well-defined procedure as interpolation, we'd probably be best advised to cast it as a general-purpose method, rather than a main program. This method would take as its arguments an ordered array of  $x_i$  values, an array of corresponding  $y_i$  (that is  $f(x_i)$ ) values, and a value  $x$ . The output would be our approximation to  $f(x)$ :

```
public class Interpolate {

    /** Given XI in strictly ascending order, YI with
     * 0 < XI.length = YI.length, and X with
     * XI[0] <= X <= XI[XI.length-1], a linear approximation
     * to a function whose value at each XI[i] is YI[i]. */

    public static double eval (double x, double[] xi, double[] yi)
    {
        For some i such that x >= xi[i] and x <= xi[i+1],
        return
            yi[i] + (yi[i+1] - yi[i]) * ((x - xi[i]) / (xi[i+1] - xi[i]));
    }
}
```

You should be able to see that the **return** expression is just a transcription of formula (1.1).

To find **i**, an obvious approach is to work our way along **xi** until we find an adjacent pair of values that bracket **x**. We can use a **for** loop, like this:

```
for (int i = 0; ; i += 1)
    if (xi[i] <= x && x <= xi[i+1])
        return ...;
```

The **return** statement has the effect of exiting the loop as well as returning a value. Since the documentation on this method requires that there be some **i** satisfying the **if** statement, we know that the function must eventually execute the **return**, if it is called properly.

This raises a more general, and very important, engineering issue: what to do when a caller violates the “contract” established by a documentation comment. In this case, Java will catch such violations indirectly. If there is no value of **i** that works properly, or if the length of **yi** is shorter than that of **xi**, a call to this method will eventually cause an *exception* when the program tries to perform an array indexing operation with an index that’s “off the end” of the array. We’ll talk about these more in §1.15. For now it suffices to say that causing (in Java terminology, *throwing*) an exception halts the program with a somewhat informative message.

### 1.10.2 Example: The Sieve of Eratosthenes

We’ve considered one way to find all primes by using a function that determines whether any particular integer is prime. There are ways to do compute primes “by the batch,” one of the simplest of which is the *sieve of Eratosthenes*. The idea behind this algorithm is to find primes one at time, and for each prime found, to immediately eliminate all multiples of the prime from any further consideration. Conceptually, we start with a sequence of integers, like this:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Starting at the left, we find the next empty box. Since it is marked ‘2’, we cross off every second box, starting from  $2^2 = 4$ , giving

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
		X		X		X		X		X		X		X		X		X		X		X		X

The process now repeats. The next empty box is marked ‘3’, so we cross off each third box, starting from  $3^2 = 9$ , giving

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
		X		X		X	X	X		X		X	X	X		X		X	X	X		X		X

Now cross off every fifth box starting from  $5^2 = 25$ , giving

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
		X		X		X	X	X		X		X	X	X		X		X	X	X		X	X	X

No further work is needed (since  $7^2$  is off the end of our sequence), and the boxes that remain empty correspond to primes.

Why does this work? Basically, it is set up in such a way that we cross off every number that is divisible by some smaller prime. Since the primes are not divisible by smaller primes, they are never crossed off. We can start crossing starting off at the square because we know that if a prime  $p$  divides a composite number  $q < p^2$ , then  $q$  will already have been crossed off. This is because  $q/p$  is a factor of  $q$  and  $q/p < p$  if  $q < p^2$ . So we crossed off  $q$  either when we processed  $q/p$  (if it is prime) or when we processed one of its (smaller) prime factors.

Java has no direct notion of “boxes that can be crossed off,” so we must find a way to *represent* these with the constructs we do have. A convenient possibility is to use an array of **boolean** values, with, say **false** meaning “not crossed off yet” (possibly prime) and **true** meaning “crossed off” (definitely composite). Here is one possible implementation:

```

/** Returns an array, p, of size N+1 such that p[k] is true iff
 * k is a prime. */
public static boolean[] primes (int n)
{
    boolean[] sieve = new boolean[n+1];
    // All entries are initially false.
    sieve[0] = sieve[1] = false;

    for (int k = 2; k*k <= n; k += 1) {
        /* If k is prime (has not been crossed off), then */
        if (! sieve[k]) {
            /* Cross off every k'th number, starting from k*k */
            for (int j = k*k; j <= n; j += k)
                sieve[j] = true;
        }
    }

    return sieve;
}

```

To use this function in `printPrimes` (page 41):

```

private static void printPrimes (int limit) {
    boolean[] isPrime = primes (limit);
    int n;
    n = 0;

```

```

    for (int L = 2; L <= U; L += 1) {
        if (isPrime[L]) {
            if (n != 0 && n % 10 == 0)
                System.out.println ();
            System.out.print (L + " ");
            n += 1;
        }
    }
}

```

### 1.10.3 Multi-dimensional Arrays

As the illustrations suggest, an array is naturally thought of as a one-dimensional sequence of things. Suppose, though, that we want to represent some sort of two-dimensional table of values—something like a matrix in linear algebra. One possibility is to make creative use of a one-dimensional array. For example, represent the matrix

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 5 & 8 & 11 \\ 3 & 7 & 11 & 15 \end{pmatrix}$$

with the array

```
int[] A = { 1, 3, 5, 7, 2, 5, 8, 11, 3, 7, 11, 15 };
```

The problem is one of inconvenience for the programmer. Instead of typical mathematical notation such as  $A_{i,j}$  for the element at row  $i$  and column  $j$ , one is forced to write `A[i * 4 + j]` instead (assuming rows and columns numbered from 0). Therefore, programming languages devoted to scientific computation (where you do this sort of thing a lot) usually have some notation specifically for multi-dimensional arrays as well.

In Java, though, we can simply rely on the fact that arrays can be arrays of any type of value, including arrays. So, our  $3 \times 4$  matrix, above, may be represented as an array of 3 4-element integer arrays:

```

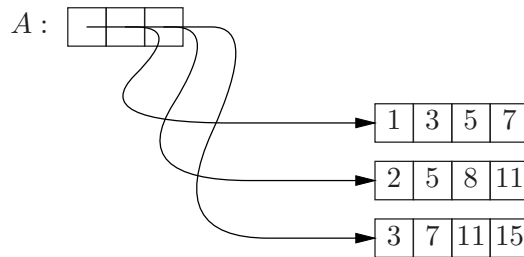
int[][] A;
A = new int[][] { new int [] { 1, 3, 5, 7 },
                  new int [] { 2, 5, 8, 11 },
                  new int [] { 3, 7, 11, 15 } };

```

or, since this is very tedious to write, we can put everything into the declaration of `A`, in which case we are allowed a little abbreviation:

```
int[][] A = { { 1, 3, 5, 7 }, { 2, 5, 8, 11 }, { 3, 7, 11, 15 } };
```

Diagrammatically,



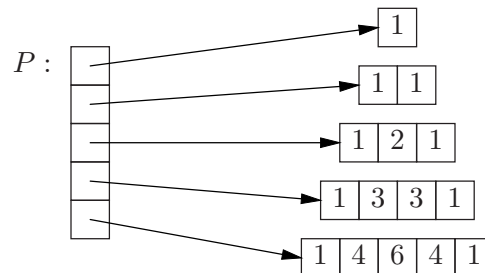
Now  $A[i]$  gives us the entire  $i^{\text{th}}$  row and  $A[i][j]$  gives us  $A_{ij}$ .

As for one-dimensional arrays, we also have the option of creating a “blank” array, and later filling it in, as in this program to create a Hilbert matrix, which is

$$\begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 \cdots \\ 1/2 & 1/3 & 1/4 & 1/5 \cdots \\ 1/3 & 1/4 & 1/5 & 1/6 \cdots \\ 1/4 & 1/5 & 1/6 & 1/7 \cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

```
public static double[][] Hilbert (int N) {
    double[][] result = new double[N][N];
    for (int i = 0; i < N; i += 1)
        for (int j = 0; j < N; j += 1)
            result[i][j] = 1.0 / (i + j + 1);
    return result;
}
```

Although nice rectangular arrays such as these are the most common, the rows are all quite independent of each other, and nothing keeps us from creating more irregular structures, such as this version of Pascal’s triangle (in which each square contains either 1 or the sum of the two squares above it):



Here’s a possible program for this purpose:

```
public static int[][] Pascal (int N) {
    // Create array of N initially null rows.
    int[][] result = new int[N][];
    for (int i = 0; i < N; i += 1) {
        // Set row #i to a row of i+1 0’s.
    }
```





```
Point A, B;
```

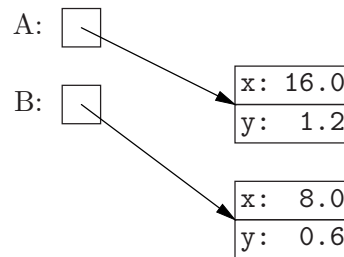
tells us that A and B can contain pointers to `Point` objects. Initially, however, A and B contain null pointers. The expression `new Point()` has as its value a new (i.e., never-before-seen) pointer to a `Point` object, so that after

```
A = new Point ();
B = new Point ();
```

A and B do point to `Point` objects. The instance variables are now available as `A.x` and `A.y`, so that after

```
A.x = 16.0; A.y = 1.2;
B.x = A.x / 2.0; B.y = A.y / 2.0;
```

we have



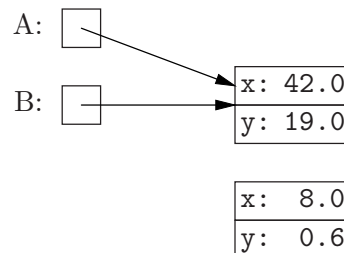
It's probably not too early to emphasize that A and B contain *pointers*, not `Point` objects. To see why this distinction matters, consider the result of

```
B = A;
B.x = 42.0; B.y = 19.0;
System.out.println ("A.x = " + A.x + ", A.y = " + A.y);
```

Here's what gets printed:

```
A.x = 42.0, A.y = 19.0
```

despite the fact that we never explicitly assigned to `A.x` or `A.y`. Here's the final situation pictorially:



As you can see, the object originally pointed to by B has not been changed (despite the assignment `B = A`), only B (the pointer variable) has changed so that it now contains the same contents (arrow) as does A. *The values of A and B are pointers, not objects.*

### 1.11.2 Instance Methods

The centrality of classes in the syntax of Java reflects an underlying philosophy of program construction, *object-based programming*, characterized by having programs organized by types of object. That is, the actual executable code is collected into methods that are themselves grouped by class (object type). From the programmer's point of view, a type of object is characterized by the operations that may be performed on it. We've seen one kind of operation: *selection* of fields (like `x` and `y` above) using `'.'` syntax. More generally, we'd like to be able to introduce arbitrarily complex operations; for this purpose we use methods.

So far, we've looked only at static methods, which correspond to ordinary sub-programs such as are found in all programming languages. Like other object-based languages, Java provides an additional kind of method—an *instance method*—that is intended to be used on objects of a class. For example, suppose that we'd like to know how far a particular `Point` is from the origin. We might first try expanding the class as follows:

```
class Point {
    double x, y;

    /** Distance from P to the origin. */
    static double dist (Point p) {
        return Math.sqrt (p.x * p.x + p.y * p.y);
    }
}
```

And now, given a point `Q`, we write `Point.dist(Q)` to get its length. However, most Java programmers would write `dist` differently:

```
/** Distance of THIS from the origin. */
double dist () {
    return Math.sqrt (this.x * this.x + this.y * this.y);
}
```

Since we don't say **static**, this version of `dist` is an *instance method*. To use it on `Q`, we write `Q.dist()`.

The change is largely “syntactic sugar.” In effect, an instance method,  $f$ , that is defined in a class named  $C$

- Has an implicit first parameter that is not listed in the parameter list. The formal type of the parameter is  $C$  and its name is **this**;
- Is called using the syntax  $E.f(\dots)$ , where  $E$  evaluates to a (non-null)  $C$ .  $E$  becomes the value of **this**, the implicit first parameter.

That is, the instance-method version of `dist` corresponds roughly to a static method declared:

```
static double dist (Point this) { ... }
```

and `Q.dist()` corresponds to `Point.dist(Q)` (however, **this** is reserved and you may not use it explicitly as a parameter name)<sup>7</sup>.

As an additional shorthand, inside the methods of `Point`, the names of instance methods and variables are automatically prefixed with ‘`this.`’ if they appear bare. So for example, we could write

```
class Point {
    double x, y;

    double dist () {
        return x*x + y*y;
    }
}
```

and the occurrences of `x` and `y` in `dist` are equivalent to `this.x` and `this.y`

### 1.11.3 Constructors

One particularly useful kind of method takes control at the time a new object is created. These are called constructors, and have their own special syntax; for our running example:

```
class Point {
    double x, y;

    /** A point whose coordinates are (x,y) */
    public Point (double x, double y) {
        this.x = x; this.y = y;
    }

    ...
}
```

The constructor is distinguished by having the same name as the class and no return type<sup>8</sup>. With this addition to `Point`, we may rewrite some previous initializations like this:

```
A = new Point (16.0, 1.2);
B = new Point (A.x / 2.0, A.y / 2.0);
```

In the absence of any explicit constructor, Java defines one with no parameters for you (which does nothing). However, now that we’ve defined a `Point` constructor,

---

<sup>7</sup>At this point, you may wonder why the designers of Java (and many previous object-based languages such as C++) chose such an annoyingly different syntax for instance methods. Your author wonders the same thing. This “worship of the first parameter” appears to be an historical accident resulting from somebody’s bright idea unfortunately taking root. What can I say? It happens.

<sup>8</sup>Again, your author finds the introduction of new syntax for this purpose to be completely unjustified. This particular piece of gratuitous irregularity is inherited from C++.

Java doesn't automatically supply one and we can no longer write `new Point()`. If we still need to do so, we can introduce another definition:

```
/** A point whose coordinates are (x,y) */
public Point (double x, double y) {
    this.x = x; this.y = y;
}

/** The point (0,0) */
public Point () { }
```

This is an example of *overloading*—the use of the same name (in this case, the same constructor name) for two different functions, distinguished by how we call them.

#### 1.11.4 Example: A Prime-Number Class

In §1.10.2, we saw how to compute batches of prime numbers, using an array as the data structure. Let's look at a typically object-based way of packaging the prime number sieve.

The main philosophical idea behind this design is that from the point of view of someone who needs to use prime numbers, the sieve is a set of prime numbers and the most important operation is a membership test: Is  $x$  in the set of prime numbers? This suggests a class that looks like this:

```
/** A finite set of prime numbers */
public class PrimeSieve {
    /** The set of prime numbers <= N. */
    public PrimeSieve (int N) { ... }

    /** A number, N, such that THIS contains exactly the primes <= N.
     * (This was the number supplied to the constructor.) */
    public int limit () { ... }

    /** Returns true iff X is in THIS. */
    public boolean contains (int x) { ... }

    ...
}
```

I have been a bit more careful than usual, and used **public** to indicate that the class, its constructor, and the `limit` and `contains` methods are intended to be used anywhere in the program.

To complete the implementation of `PrimeSieve`, we copy in the pieces from §1.10.2 to give the result in Figure 1.2. This time, the actual sieve representation (the array) is declared to be **private**, and therefore not directly available outside of `PrimeSieve`. Since the only code that can touch the instance variable `sieve` is what's written here, we can be sure that as long as the program text in `PrimeSieve` works, these methods will give the answers their comments call for.

Finally, here's how we modify `printPrimes` to use this new class:

```
private static void printPrimes (int limit) {
    PrimeSieve primes = new PrimeSieve (limit);
    int n;
    n = 0;
    for (int L = 2; L <= U; L += 1) {
        if (primes.contains (L)) {
            if (n != 0 && n % 10 == 0)
                System.out.println ();
            System.out.print (L + " ");
            n += 1;
        }
    }
}
```

## 1.12 Interfaces

Let's break up the `printPrimes` method in §1.11.4 into a piece that knows about primes and one that knows about printing (on page 53):

```
private static void printPrimes (int limit) {
    printSet (new PrimeSieve (limit));
}

static void printSet (PrimeSieve set) {
    int n;
    n = 0;
    for (int L = 0; L <= set.limit (); L += 1) {
        if (set.contains (L)) {
            if (n != 0 && n % 10 == 0)
                System.out.println ();
            System.out.print (L + " ");
            n += 1;
        }
    }
}
```

I've modified the **for** loop in `printSet` so that it no longer assumes that the smallest prime is 2. As a result, the body of `printSet` would correctly display any set of non-negative numbers that provided `limit` and `contains` methods.

Java allows us to capture this fact by defining an *interface* that describes simple sets of non-negative integers:

```
/** A finite set of prime numbers */
public class PrimeSieve {
    /** The set of prime numbers <= N. Requires N >= 0. */
    public PrimeSieve (int N) {
        sieve = new boolean[N+1];
        for (int i = 2; i <= n; i += 1)
            sieve[i] = true;
        for (int k = 2; k*k <= n; k += 1) {
            if (sieve[k]) {
                for (int j = k*k; j <= n; j += k)
                    sieve[j] = false;
            }
        }
    }

    /** A number, N, such that THIS contains exactly the primes <= N.
     * (This was the number supplied to the constructor.) */
    public int limit () { return sieve.length - 1; }

    /** Returns true iff X is in THIS. */
    public boolean contains (int x) {
        if (x < 0 || x >= sieve.length)
            return false;
        return sieve[x];
    }

    private boolean[] sieve;
}
```

Figure 1.2: The Sieve of Eratosthenes as a Java class.

```

/** A bounded set of natural numbers. */
public interface FiniteNaturalSet {
    /** A number, N, such that all numbers in THIS are <= N. */
    int limit ();

    /** Returns true iff X is in THIS. */
    boolean contains (int x);
}

```

There are no bodies for these methods: `FiniteNaturalSet` defines only what operations are available, not how they are to be done. Likewise, there is no constructor because this interface does not define a particular kind of set, with a particular representation, and there is no universal set of fields and initializations of those fields that must be used by all possible objects that supply these two methods. With this interface, we may now write

```

static void printSet (FiniteNaturalSet set) {
    ...
    for (int L = 0; L <= set.limit (); L += 1) {
        if (set.contains (L)) {
            ...
        }
    }
}

```

This will work for any `FiniteNaturalSet`, but to make it work again for our original application, we have to indicate that a `PrimeSieve` *is a* kind of `FiniteNaturalSet`, which we do like this:

```

public class PrimeSieve implements FiniteNaturalSet {
    The rest of PrimeSieve is as before
}

```

(We also say that `PrimeSieve` is a *subtype* of `FiniteNaturalSet`.) So basically, we're back where we were before except that should we need to print other sets of natural numbers, we will not have to write new versions of `printSet`. Instead, `printSet` will, in effect, adapt itself to any variety of `FiniteNaturalSet` object we pass to it, calling the `contains` and `limit` functions that were defined for that kind of set.

For example, here's a class that defines a set consisting of all perfect squares within some range<sup>9</sup>:

```

public class SquareSet implements FiniteNaturalSet {
    /** The set consisting of { LOW^2, (LOW+1)^2, ..., HIGH^2 },
     *  where 0 <= LOW, HIGH < square root of 231 */
    public SquareSet (int low, int high) {

```

---

<sup>9</sup>You might have to puzzle over how `contains` works here. Go ahead; it'll be good for you.

```

        this.low = low; this.high = high;
    }

    public int limit () { return high*high; }

    public boolean contains (int x) {
        if (x < low*low || x > high*high)
            return false;
        // Note: (int) E, where E is a double, truncates E (rounds toward
        // 0) to an integer value.
        int approx_root = (int) Math.sqrt (x);
        return (approx_root*approx_root == x ||
                (approx_root+1)*(approx_root+1) == x);
    }

    private int low, high;
}

```

so that now,

```
printSet (new SquareSet (1, 100))
```

prints all the squares up to  $100^2$ . As you can see `SquareSet` is very different from `PrimeSieve`, even though both are `FiniteNaturalSets`. In particular, `SquareSet` does not compute all the squares in its constructor, but tests each argument to `contains` separately. All of this detail is irrelevant to `printSet`, and is hidden behind the interface.

## 1.13 Inheritance

As it happens, interfaces are just special cases of classes, which actually provide a more powerful facility. Specifically, interfaces are restricted not to contain instance variables or implementations of methods. We could instead have written

```

public abstract class FiniteNaturalSet {
    /** A number, N, such that all numbers in THIS are <= N. */
    public abstract int limit ();

    /** Returns true iff X is in THIS. */
    public abstract boolean contains (int x);
}

```

and then

```
public class PrimeSieve extends FiniteNaturalSet { ... }
```

(note the different keyword). As was the case for **implements**, a `PrimeSieve` is a `FiniteNaturalSet` by virtue of its declaration.



The keyword **abstract** roughly means “not completely implemented here.” An unimplemented method may not be called, and to enforce this, an object of an abstract class may not be created (i.e., with **new**). This particular class must be abstract because it contains abstract (bodiless) methods. In effect, the **interface** keyword made all methods implicitly abstract and public. Unfortunately, it is not quite that simple: one can *implement* any number of interfaces, but can *extend* only one class.

Let’s consider another example. Suppose that you were writing a program that dealt with geometrical figures in a plane—triangles, rhombi, circles, and the like. We anticipate that we’ll need things like lists or sets that can contain any kind of figure, so we’ll first need a general type that encompasses all figures. Its methods should be things that make sense for any figure. The class shown in Figure 1.3 (page 58) is one possibility. I’ve crammed quite a few features into it, so it warrants a bit of close study.

First, we see that there are two constructors, each of which constructs a **Figure** at a specific reference point. Constructor (2) simply creates a new reference **Point** and passes it to constructor (1); that’s the meaning of **this(...)** in the first line of its body—“the other constructor whose only argument is a single **Point**.”

Next we see that some of the methods—**type**, **location**, **translate**, and one **contains** method (4)—do have implementations. That’s because we can write a sensible implementation for them that applies to all **Figures**. On the other hand, we have no general way to compute the area or extent of a **Figure**, based on the information given, so **area**, and one **contains** method are abstract. Once we know how to compute **contains(x,y)**, however, we can easily compute **contains(p)**, so the latter method *is* implemented. But how can it work if it calls an unimplemented method? Wait and see.

Now let’s define a particular kind of **Figure**: a rectangle.

```
public class Rectangle extends Figure {
    /** A Rectangle with sides parallel to the axes, width W,
     * height H, and lower-left corner X0, Y0. */
    public Rectangle (double x0, double y0, double w, double h) { // (7)
        super (x0, y0);
        this.w = w; this.h = h;
    }

    public String type () { // (8)
        return "rectangle";
    }

    /** The area enclosed by THIS. */
    public double area () { // (9)
        return w * h;
    }
}
```

```

/** A closed figure in the plane. Every Figure has a "reference
 * point" within it, which serves as the location of the Figure. */
public abstract class Figure {
    /** A Figure with reference point P0. */
    public Figure (Point p0) {                // (1)
        this.p0 = p0;
    }

    public Figure (double x0, double y0) {    // (2)
        this (new Point (x0, y0));
    }

    /** The generic name for THIS's type of figure
     * (e.g., "circle"). This is the default implementation. */
    public String type () {                  // (3)
        return "2d figure";
    }

    /** The area enclosed by THIS. */
    public abstract double area ();

    /** True iff (X,Y) is inside THIS. */
    public abstract boolean contains (double x, double y);

    /** True iff P is inside THIS. */
    public boolean contains (Point p) {      // (4)
        return contains (p.x, p.y);
    }

    /** The reference point for THIS. */
    public Point location () { return p0; }  // (5)

    /** Translate (move) THIS by DX in the x direction and DY in the
     * y direction. */
    public void translate (double dx, double dy) { // (6)
        p0.x += dx; p0.y += dy;
    }

    /** A rough printable description of THIS. */
    public String toString () {
        return type () + " at (" + p0.x + ", " + p0.y + ")";
    }

    private Point p0;
}

```

Figure 1.3: The abstract class representing all **Figures**. Its constructors deal only with its location.

```

/* Continuation of Rectangle */

/** True iff (X,Y) is inside THIS. */
public boolean contains (double x, double y) { // (10)
    Point p = location ();
    return x >= p.x && x <= p.x+w
           && y >= p.y && y <= p.y+h;
}

private double w, h;
}

```

This class is not abstract—all its methods are implemented. Because it extends **Figure**, it *inherits* all the fields and methods that **Figure** has. That is, the implementations of methods (4), (5), and (6) are effectively carried over into **Rectangle** (so if **R** is declared

```
Rectangle R = new Rectangle (1.0, 2.0, 4.0, 3.0);
```

then **R.location ()** gives its location, the point (1.0,2.0)).

Since a **Rectangle** is a **Figure**, the constructor (7) must first call the constructor for **Figure**, which is the meaning of the “call” to **super** as the first line of its body. Next, the **Rectangle** constructor initializes the instance variables that are specific to it: **w** and **h**.

The **Rectangle** class provides implementations for **area** and the two-argument **contains** method. The call **R.contains(p)** (on method (4)) will call the **contains** method (10). A call on **R.toString ()** will return the result

```
rectangle at (1.0, 2.0)
```

That is, even though **toString** is defined in **Figure** at (3), this call will use the **type** function that is defined in **Rectangle** at (8). We say that definition (8) *overrides* definition (3). (Likewise, (9) and (10) override the matching definitions in **Figure**, but since the latter don’t have bodies, there is nothing terribly surprising in that fact.)

What this all illustrates is that

A call on an instance (i.e., non-static) method  $A.f(\dots)$  chooses which matching definition of  $f$  to use depending on the *current value* of  $A$ .

So if  $A$ ’s value is a pointer to a **Rectangle**, then the definition of  $f$  in the **Rectangle** class prevails (if there is no such method, we use the inherited definition).

**Question:** To test that you understand this rule, suppose we added the following method to the **Figure** class:

```

static void prt (Figure f) {
    System.out.println (f.toString ());
}

```

If we now execute the call

```
Figure.prt (new Rectangle (1.0, 2.0, 4.0, 3.0));
```

what happens?

**Answer:** Despite the fact that `prt` is defined in `Figure` and despite the fact that its formal parameter, `f`, is declared to be a `Figure`, this call prints

```
rectangle at (1.0, 2.0)
```

just as before. It is the *value* of `f` (a pointer to a rectangle) that matters, not its declared type. Yes, `f` points to a `Figure`, but it is a particular kind of `Figure`—specifically a `Rectangle`.

## 1.14 Packages and Access Control

The keywords **public**, **private**, and **protected** are features of the Java language that support what parts of your program have access to what instance variables, classes, and methods. They add no programming power, but are intended instead to support common notions of disciplined program design. The underlying idea is simple. The designer of a particular class in effect makes a “contract” with the users (*clients*) of the class that guarantees the behavior of particular methods. The clients, for their part, must supply legal arguments to these methods, and must keep their hands off the internals (methods and variables not mentioned in the contract).

The **public** keyword is a syntactic marker indicating classes and members that are part of the contract. The other possible keywords (and the absence of a keyword) indicate classes and members that are not part of the contract. Unfortunately, there are various degrees of “confidentiality,” requiring more than one possible non-public access level. The obvious kind is indicated by the keyword **private**: a private member is accessible only within the actual text of a class definition; no outside class knows about such members. When possible, make your non-public members private. Sometimes, however, more than one class must participate in implementing a feature, and perfect privacy is impractical.

For this purpose, Java allows us to group classes into *packages*, and to restrict some information to the package. The standard Java library contains quite a few packages. One of them, called `java.lang`, contains classes that are basic to the language (like `String`). The declaration

```
package numbers;
```

```
public class PrimeSieve extends FiniteNaturalSet { ...
```

as the first statement in our previous example tells us that `PrimeSieve` is in the package `numbers`, so that its full name (when referenced from outside the package) is `numbers.PrimeSieve`. Many programs tend to have no package declaration, which simply puts them in the *anonymous package*.

To look at a more generic case:

```

package demo;

public class A {                // (0)
    public A (int x) { ... }    // (1)
    A () { ... }                // (2)
    void f () { ... }           // (3)
    protected g () { ... }      // (4)
    private h () { ... }        // (5)
}

class B {                        // (6)
    ...
}

```

This tells us that declarations (2), (3), and (6) are *package private*: they may be used only in package `demo`. As a result, in package `demo`, one can write `new A()`, but outside one can only write `new A(3)`. The restrictions on protected declaration (4) are a little looser, classes that extend `demo.A` may also call `g`, even if those classes are not in package `demo`.

You will find it annoying to denote classes by their full *qualified* names all the time. This is especially true of the Java standard library classes—`java.util.Map` is not a particularly enlightening elaboration of `Map`, for example. There are four useful devices to alleviate this annoyance and let you use an unqualified or *simple* name instead of the fully qualified name:

- The declaration

```
import java.util.Map;
```

at the beginning of a program (after any **package** declaration, and before any class declaration) means that the simple name `Map` may be substituted for `java.util.Map` in this file.

- The declaration

```
import java.util.*;
```

in effect imports all the classes in the package `java.util`.

- Finally, all programs have an implicit `import java.lang.*`; at the beginning. Therefore, you can always write `String` rather than `java.lang.String` and `Object` rather than `java.lang.Object`.
- Within the text of a definition in a package, names of other classes and interfaces in the package don't have to be qualified.

## 1.15 Handling Exceptional Cases

A sad fact of software life is that inordinate amounts of programmers' time is spent writing program text whose sole purpose is to detect erroneous conditions. This fact is reflected in many modern programming languages, in the form of features that are intended to detect erroneous uses of the language, and to report such errors or refer them to parts of the program that can deal with them. In Java, this reporting device is called *throwing an exception*.

### 1.15.1 Built-in Exceptions

There are any number of things that one's program is Not Supposed to Do. For example, the following is wrong:

```
class Bad {
    static void blowUp () {
        String s = null;
        if (s.equals(""))           // (??)
            System.out.println ("Oops!");
    }

    public static void main (String[] args) {
        blowUp ();
    }
}
```

The problem is that you have tried to call the `equals` method (an instance method) on a “non-instance”—the null pointer doesn't point at anything. Java responds by throwing an exception when the condition at (??) is evaluated. Assuming you are not expecting it, what probably happens is that your entire program grinds to a halt with a message such as this:

```
Exception in thread "main" java.lang.NullPointerException
    at Bad.blowUp(Bad.java:4)
    at Bad.main(Bad.java:9)
```

which tells us where the error occurred (in the method `Bad.blowUp` in file `Bad.java` at line 4) and how we got to that point (we were called from `Bad.main` at line 9 of the same file). This message (called a *back trace* or *stack trace*) also gives us the *exception type*: `NullPointerException`, which is suggestively named to tell us what we did wrong.

You'll get any number of similar messages for other programming errors, such as attempting to use an array index that is out of bounds, or attempting to do an integer division by 0.

### 1.15.2 What Exactly is an Exception?

It is typical in object-oriented programming that nouns used informally to describe the important concepts or entities in one's program are turned into classes of object.

Accordingly, “exception” in Java is actually a class of object known as **Throwable** (full name `java.lang.Throwable`, see §1.14), which has numerous subtypes that extend it, each one representing a particular kind of exceptional condition. These exceptions are organized into three broad categories under three of the subtypes: **Error** intended for calamitous program-stopping problems such as running out of memory; **RuntimeException** for exceptions caused by built-in operations (e.g., **NullPointerException**) or other programmer errors; and **Exception** for conditions that are not necessarily programmer errors (for example, that may be caused by I/O errors or other conditions that arise outside the program).

### 1.15.3 Throwing Exceptions Explicitly

The **throw** statement throws an exception. Here’s a simple extension to one of our previous examples:

```
/** True iff X is divisible by any positive number >=K and < X,
 *  given K > 1. */
private static boolean isDivisible (int x, int k) {
    if (k <= 1)
        throw new IllegalArgumentException ();
    ...
}
```

**IllegalArgumentException** is conveniently supplied as part of the Java library to allow one to signal, well, illegal arguments. As you can see, the **throw** command takes a single operand, a pointer to some kind of **Throwable** object, which in this case (as is usually done, in fact) we create on the spot with **new**.

### 1.15.4 Catching Exceptions

Sometimes, you expect to receive an exception—for example many input/output operations carry the possibility of an externally caused error. Robust programs are prepared for such events. Java provides a way to say “I know that exceptions of type *X* can occur during the execution of statements *S*. If one does, perform statements *E*.” For example, to starting read a file Java, one often does something like this:

```
Reader openFile (String fileName) {
    try {
        return new FileReader (fileName);
    } catch (FileNotFoundException e) {
        System.err.println ("Could not open file " + fileName);
        return null;
    }
}
```

to indicate that the program should print an error message if the desired file cannot be found.





## Chapter 2

# Describing a Programming Language

A *programming language* is a notation for describing computations or processes. The term “language” here is technically correct, but a little misleading. A programming language is enormously simpler than any human (or *natural*) language, for the simple reason that for it to be useful, somebody has to write a program that converts programs written in that language into something that runs on a computer. The more complex the language, the harder this task. So, programming “languages” are sufficiently simple that, in contrast to English, one can actually write down complete and precise descriptions of them (at least in principle).

### 2.1 Dynamic and Static Properties

Over the years, we’ve developed standard ways to describe programming languages, using a combination of formal notations for some parts and semi-formal prose descriptions for other parts. A typical language description consists of a *core*—the general rules that determine what the legal (or *well-formed*) programs are and what they do—plus *libraries* of definitions for standard, useful program components. The description of the core, in turn, typically divides into *static* and *dynamic* properties.

**Static properties.** In computer science, we use the term *static* in several senses. A static language property is purely a property of the *text* of a program, independent of any particular set of *data* that might be input to that program. Together, the static properties determine which programs are *well-formed*; we only bother to define the effects of well-formed programs. Traditionally, we divide static properties into *lexical structure*, (*context-free*) *syntax*, and *static semantics*.

- *Lexical Structure* refers to the alphabet from which a program may be formed and to the smallest “words” that it is convenient to define (which are called *tokens*, *terminals*, or *lexemes*.) I put “word” in quotes because tokens can be far more varied than what we usually call words. Thus, numerals, identifiers, punctuation marks, and quoted strings are usually identified as tokens.

Rules about what comments look like, where spaces are needed, and what significance ends of lines have are also considered lexical concerns.

- *Context-free Syntax* (or *grammar*) refers roughly to how tokens may be put together, ignoring their meanings. We call a “sentence” such as “The a; walk bird” syntactically incorrect or ungrammatical. On the other hand, the sentence “Colorless green ideas sleep furiously” is grammatical, although meaningless<sup>1</sup>.
- *Static Semantics* refers to other rules that determine the meaningfulness of a program (“semantics” means “meaning”). For example, the rule in Java that all identifiers used in a program must have a definition is a static semantic rule.

The boundaries between these three may seem unclear, and with good reason: they are rather arbitrary. We distinguish them simply because there happen to be convenient notations for describing lexical structure and context-free syntax. There is considerable overlap between the categories, and sometimes it is a matter of taste which notation one uses for a particular part of a programming language’s description.

**Dynamic Properties.** The *dynamic semantics* of a program refers to its meaning as a computation—what it does when it is executed (assuming, of course, that the rules governing static properties of the programming language tell us that it is well-formed). For example, the fact that `x+y` fetches the current values of variables `x` and `y` and yields their sum is part of the dynamic semantics of that expression. In general, the term *dynamic* in the context of programming languages refers to properties that change (or are only determined) during the execution of a program. For example, the fact that the expression `x>y` yields a boolean (true/false) value is a static property of the expression, while the actual value of the expression during a particular evaluation is a dynamic property. As usual, the boundaries between static and dynamic can be unclear. For example, the evaluation of `3>4` follows the same rules for ‘>’ as does the evaluation of `x>y`, but its value is always the same (**false**). Should we call the value of `3>4` a static or dynamic property? We’ll generally follow the sensible course of simply ignoring such questions.

**Libraries.** In one sense, a program consists of a set of definitions, all building on one another, one of which may be identified as the “main program.” There have to be some primitive definitions with which to start. The core defines some of these primitive definitions, and others come from *libraries* of definitions. Typically, there will be set of libraries—referred to as the *standard (runtime) library* or sometimes as the *standard prologue*—that “comes with” the programming language, and is present in every implementation. One could consider it part of the core, except that it is typically described as a set of declarations of the sort that any program could contain.

---

<sup>1</sup>This sentence is a famous example due to Noam Chomsky.

## 2.2 Describing Lexical Structure and Syntax

The original description of the Algol 60 language<sup>2</sup> is the model on which many other “official” descriptions of programming languages have been based. It introduced a novel notation (adapted, really, from linguistics) for describing the (context-free) syntax of a language that came to be known as Backus-Naur Form or Backus Normal Form after its inventors (the usual abbreviation, *BNF*, works for both)<sup>3</sup>. In this book, we’ll use a somewhat extended version of BNF notation.

The basic idea is simple. A BNF description is a sequence of definitions of what are called *syntactic variables*, or *nonterminals*. Each such variable stands for a set of possible phrases. One distinguished variable (the *start symbol*) stands for the set of all grammatical phrases in the language we are trying to describe. For example<sup>4</sup>:

*Sentence*: *NounPhrase VerbPhrase*  
*NounPhrase*: *Adjective NounPhrase*  
*NounPhrase*: *Adjective PluralNoun*  
*VerbPhrase*: *PluralVerb Adverb<sub>opt</sub>*  
*Adjective*: laughing  
*Adjective*: little  
*SingularNoun*: boys  
*SingularNoun*: girls  
*SingularVerb*: ran  
*Adverb*: quickly

Read the first definition, for example as “A *Sentence* may be formed from a *NounPhrase* followed by a *VerbPhrase*.” We read ‘:’ as “may be” as opposed to “is,” because as you can see, there can be several ways to form certain kinds of phrase. The second definition illustrates *recursion*: a *NounPhrase* may be formed from an *Adjective* followed by a (smaller) *NounPhrase*. The recursion stops when we use the second definition of *NounPhrase*. The definition of *VerbPhrase* illustrates a useful piece of notation: the subscript ‘*opt*’ to indicate an optional part of the definition. Equivalently, we could have written

*VerbPhrase*: *PluralVerb Adverb*  
*VerbPhrase*: *PluralVerb*

Each of the terms that does not appear to the left of a ‘:’ is a *terminal symbol*. Our convention will be that non-italicized terminals “stand for themselves”—thus, “laughing little boys ran” is a phrase generated by this grammar. As you can see, we don’t mention *lexical* details such as blanks when giving grammar rules; we’ll

<sup>2</sup>P. Nauer, ed., “Report on the Algorithmic Language Algol 60,” *Comm ACM*, 6(1), 1963, pp. 1–17.

<sup>3</sup>In notations where the same thing can be written in many different ways, a *normal form* for some expression is supposed to be a particular choice among these different ways that is somehow unique. For example, the expressions  $x - 3$ ,  $x + (-3)$ ,  $-3 + x$ , and  $1 \cdot x - 5 + 2$  all refer to the same value for any  $x$ . A normal form might be “ $a \cdot x + b$ , where  $a$  and  $b$  are constants.” There is only one way to write the expression that way:  $1 \cdot x + (-3)$ . BNF does *not* have this property, so of the two names, I prefer Backus-Naur Form.

<sup>4</sup>This example is adapted from Hopcroft and Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969.

deal with those details separately.

There are several pieces of notational trickery we'll use to shorten definitions a bit. Multiple definitions of the same nonterminal may be collected together and written either like this:

*NounPhrase*: *Adjective NounPhrase* | *Adjective PluralNoun*

(that is, read '|' as "or"). With longer definitions, we'll usually use the following style:

*NounPhrase*:  
     *Adjective NounPhrase*  
     *Adjective PluralNoun*

The effect of our definition of *NounPhrase* is to define it to be a sequence of one or more *Adjectives* followed by a *PluralNoun*. We'll sometimes use a traditional shorthand for this and write its definition as simply

*NounPhrase*: *Adjective*<sup>+</sup> *PluralNoun*

The raised '+' means "one or more." Similarly, there is a related notation that we might use to define another kind of *NounPhrase* in which the *Adjectives* are optional:

*SimpleNounPhrase*: *Adjective*\* *PluralNoun*

The asterisk here (called the *Kleene star*) means "zero or more."

Two other common cases are the "list of one or more *Xs* separated by *Ys*" and the "list of one or more *Xs* separated by *Ys*," which one can write as in these examples:

*ExpressionList*: *Expression* | *ExpressionList* , *Expression*  
*ParameterList*: *ExpressionList*<sub>opt</sub>

Rather than defining new nonterminals, however, we'll allow a shorthand that replaces *ExpressionList* and *ParameterList* with

*Expression*<sup>+</sup>,  
*Expression*<sup>\*</sup>,

respectively.

**Continuing lines.** Sometimes a definition gets too long for a line. Our convention here is to indent continuation lines to indicate that they are *not* alternatives. For example, this is one continued alternative, not two:

*BasicClassDeclaration*:  
     **class** *SimpleName* *Extends*<sub>opt</sub> *Implements*<sub>opt</sub>  
         { *ClassBodyDeclaration*<sup>\*</sup> }

## Chapter 3

# Lexical Basics

### 3.1 Layout: Whitespace and Comments

Like many modern languages, the text of Java programs is written in *free format*, meaning that extra *whitespace*—a general term for blanks, tab characters, and ends-of-line—is ignored. The following four alternative ways of writing `x=x*(y+7)`, for example, are all legal and equivalent (although I’d say that the last three are distinctly lacking in aesthetic merit):

<code>x = x * (y+7);</code>		<code>x = x</code>		<code>x= x* (y+ 7);</code>		<code>x=x*(y+7);</code>
		<code>* ( y + 7 )</code>				
		<code>;</code>				

Whitespace *is* needed whenever an identifier, keyword, or numeral is immediately followed by another identifier, keyword, or numeral—you must write “`int x`” and not “`intx`.”

A *comment* is a piece of text intended mostly for the (human) programmer, adding nothing to the meaning of a program. In effect, a comment is just a kind of whitespace. In Java, there are two kinds of comment, illustrated here:

```
int x; // This kind starts with // and continues to the end
      // of the line.
int y; /* This kind starts with /* and continues for any
      * number of lines, up to the next */
```

There is a special form of the second kind of comment that, while still not affecting the meaning of programs, has significance to certain other tools. It is called a *documentation comment*, defined as any comment that starts with ‘`/**`’. You’ll see plenty of examples of its use in pages to come.



## Chapter 4

# Values, Types, and Containers

A programming language is a notation for describing the manipulation of some set of conceptual entities: numbers, strings, variables, and functions, among others. A description of this set of entities—of what we call the *semantic domain* of the language—is therefore central to the description of a programming language. To experienced programmers, the notation itself—the syntax of the language—is largely a convenient veneer; the semantic domain is what they are really talking about. They think “behind the syntax” to the effects of their programs on the denizens of the semantic domains, or as the Cornell computer scientist David Gries puts it, they program *into* a programming language rather than *in* it.

To set the stage for what is to come, this chapter is devoted to developing a model for Java’s semantic domain, one that is actually applicable to a large range of programming languages. Our model consists of the following components:

**Values** are “what data are made of.” They include, among other things, integers, characters, booleans (true and false), and pointers. Values, as I use the term, are *immutable*; they never change.

**Containers** contain values and other containers. Their contents (or *state*) can vary over time as a result of the execution of a program. Among other things, I use the term to include what are elsewhere called *objects*. Containers may be *simple*, meaning that they contain a single value, or *structured*, meaning that they contain other containers, which are identified by names or indices. A container is *named* if there is some label or identifier a program can use to refer to it; otherwise it is *anonymous*. In Java, for example, local variables, parameters, and fields are named, while objects created by **new** are anonymous.

**Types** are, in effect, tags that are stuck on values and containers like Post-it<sup>tm</sup> notes. Every value has such a type, and in Java, so does every container. Types on containers determine the types of values they may contain.

**Environments** are special containers used by the programming language for its local and global named variables.

## 4.1 Values and Containers

One of the first things you’ll find in an official specification of a programming language is a description of the primitive values supported by that language. In Java, for example, you’ll find seven kinds of number (types **byte**, **char**, **short**, **int**, **long**, **float**, and **double**), true/false values (type **boolean**), and pointers. In C and C++, you will also find functions (there are functions in Java, too, but the language doesn’t treat them as it does other values), and in Scheme, you will find rational numbers and symbols.

The common features of all values in our model are that they have types (see §4.2) and they are *immutable*; that is, they are changeless quantities. We may loosely speak of “changing the value of *x*” when we do an assignment such as ‘*x* = 42’ (or ‘(set! *x* 42)’) but under our model what really happens here is that *x* denotes a *container*, and these assignments remove the previous value from the container and deposit a new one. At first, this may seem to be a confusing, pedantic distinction, but you should come to see its importance, especially when dealing with pointers.

### 4.1.1 Containers and Names

A *container* is something that can contain values and other containers. Any container may either be *labeled* (or *named*)—that is, have some kind of name or label attached to it—or *anonymous*. A container may be simple or structured. A *simple container*, represented in my diagrams as a plain rectangular box, contains a single value. A *structured container* contains other containers, each with some kind of label; it is represented in diagrams by nested boxes, with various abbreviations. The full diagrammatic form of a structured container consists of a large container box containing zero or more smaller containers<sup>1</sup>, each with a *label* or *name*, as in Figure 4.1a. Figures 4.1b–e show various alternative depictions that I’ll also use. The inner containers are known as *components*, *elements* (chiefly in arrays), *fields*, or *members*.

An *array* is a kind of container in which the labels on the elements are themselves values in the programming language—typically integers or tuples of integers. Figure 4.2 shows various alternative depictions of a sample array whose elements are labeled by integers and whose elements contain numbers.

### 4.1.2 Pointers

A *pointer* (also known as a *reference*<sup>2</sup>) is a value that designates a container. When I draw diagrams of data structures, I will use rectangular boxes to represent containers

<sup>1</sup>The case of a structured container with no containers inside it is a bit unusual, I admit, but it does occur.

<sup>2</sup>For some reason, numerous Java enthusiasts are under the impression that there is some well-defined distinction between “references” and “pointers” and actually attempt to write helpful explanations for newcomers in which they assume that sentences like “Java has *references*, not *pointers*” actually convey some useful meaning. They don’t. The terms are synonyms.



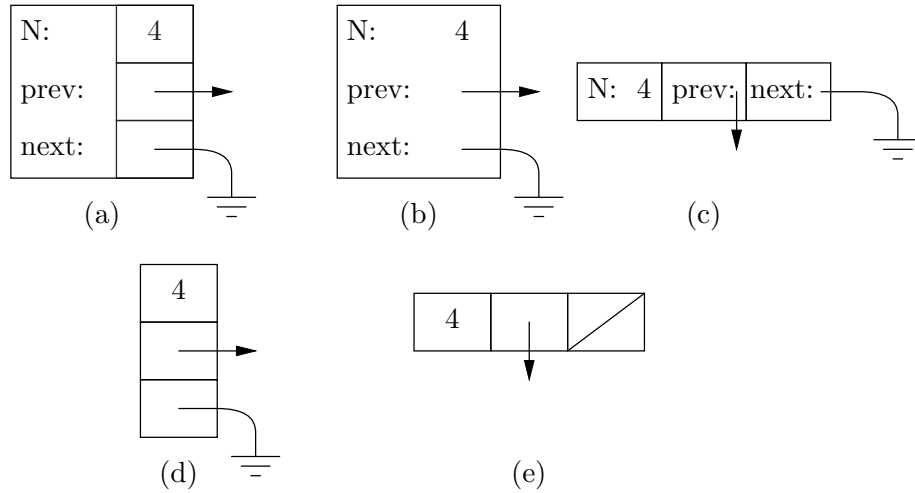


Figure 4.1: A structured container, depicted in several different ways. Diagrams (d) and (e) assume that the labels are known from context.

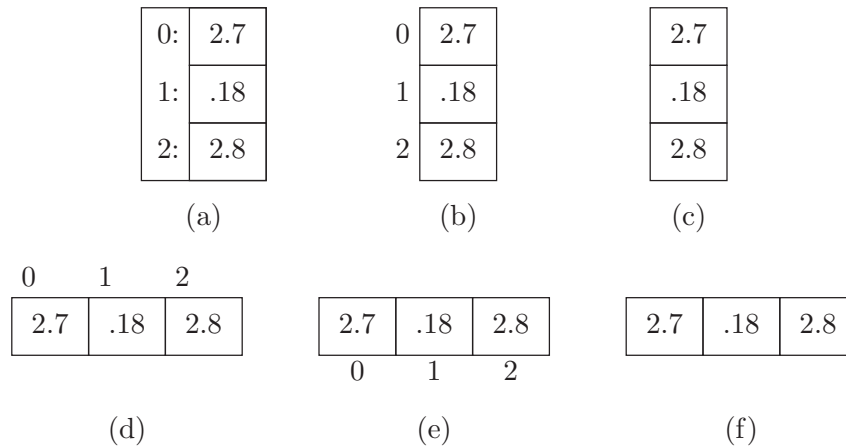


Figure 4.2: Various depictions of one-dimensional array objects. The full diagram, (a), is included for completeness; it is generally not used for arrays. The diagrams without indices, (c) and (f), assume that the indices are known from context or are unimportant.

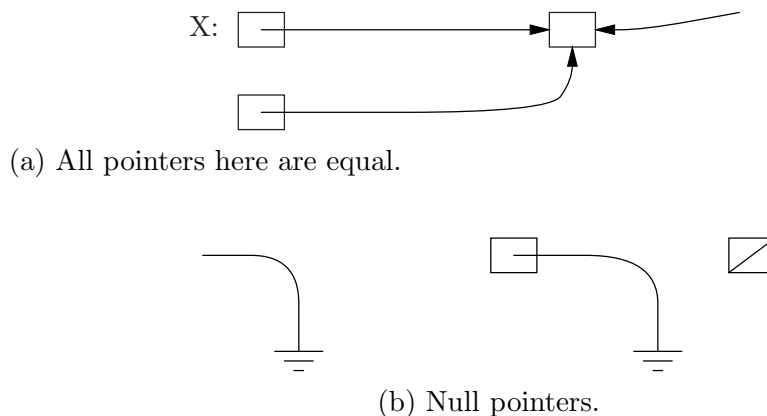


Figure 4.3: Diagrammatic representations of pointers.

and arrows to represent pointers. Two pointer values are the same if they point to the same container. For example, all of the arrows in Figure 4.3a represent equal pointer values. As shown there, we indicate that a container contains a certain pointer value by drawing the pointer's tail inside the container. The operation of following a pointer value to the container at its head (i.e., its point) in order to extract or store a value is called *dereferencing* the pointer, and the pointed-to container is the *referent* of the pointer.

Certain pointer values are known as *null pointers*, and point at nothing. In diagrams, I will represent them with the electrical symbol for ground, or use a box with a diagonal line through it to indicate a container whose value is a null pointer. Figure 4.3b illustrates these conventions with a “free-floating” null pointer value and two containers with a null pointer value. Null pointers have no referents; dereferencing null pointers is undefined, and generally erroneous.

**Value or Object?** Sometimes, it is not entirely clear how best to apply the model to a certain programming language. For example, we model a pair in Scheme as an object containing two components (`car` and `cdr`). The components of the pair have values, but does the pair *as a whole* have a value? Likewise, can we talk about the value in the arrays in Figure 4.2, or only about the values in the individual elements? The answer is a firm “that depends.” We are free to say that the container in Figure 4.2a has the value  $\langle 2.7, 0.18, 2.8 \rangle$ , and that assigning, say, 0 to the first element of the array replaces its *entire* contents with the value  $\langle 0, 0.18, 2.8 \rangle$ . In a programming language with a lot of functions that deal with entire arrays, this would be useful. To describe Java, however, we don't happen to need the concept of “the value of an array object.”

## 4.2 Types

The term “type” has numerous meanings. One may say that a type is a set of values (e.g., “the type `int` is the set of all values between  $-2^{31}$  and  $2^{31} - 1$ , inclusive.”).

Or we may say that a type is a programming language construct that defines a set of values and the operations on them. For the purposes of this model, however, I'm just going to assume that a type is a sort of *tag* that is attached to values and (possibly) containers. Every value has a unique type. This model does *not* necessarily reflect reality directly. For example, in typical Java implementations, the value representing the character 'A' is indistinguishable from the integer value 65 of type **short**. These implementations actually use other means to distinguish the two than putting some kind of marker on the values. For us programmers, however, this is usually an invisible detail.

Any given programming language provides some particular set of these type tags. Most provide a way for the programmer to introduce new ones. Few programming languages, however, provide a direct way to look at the tag on a value (for various reasons, among them the fact that it might not really be there!).

#### 4.2.1 Static vs. dynamic types

When *containers* have tags (they don't have to; in Scheme, for example, they generally don't), these tags generally determine the possible values that may be contained. In the simplest case, a container labeled with type *T* may only contain values of type *T*. In Java (and C, C++, FORTRAN, and numerous other languages), this is the case for all the numeric types. If you want to store a value of type **short** into a container of type **int**, then you must first *coerce* (a technical term, meaning *convert*) the **short** into an **int**. As it happens, that particular operation is often merely notional; it doesn't require any machine instructions to perform, but we can still talk that way.

In more complex cases, the type tag on a container may indicate that the values it contains can have one of a whole set of possible types. In this case, we say that the allowable types on values are *subtypes* of the container's type. In Java, for example, if the definition of class **Q** contains the clause "**extends P**" or "**implements P**," then **Q** is a subtype of **P**; a container tagged to contain pointers to objects of type **P** may contain pointers to objects of type **Q**. The subtype relation is transitive: any subtype of **Q** is also a subtype of **P**. As a special case, any type is a subtype of itself; we say that one type is a *proper subtype* of another to mean that it is an unequal subtype.

If type *C* is a subtype of type *P*, and *V* is a value whose type tag is *C*, we say that "*V is a P*" or "*V is an instance of P*." Unfortunately, this terminology makes it a little difficult to say that *V* "really is a" *P* and not one of its proper subtypes, so in this class I'll say that "the type of *V is exactly P*" when I want to say that.

In Java, all objects created by **new** are anonymous. If **P** is a class, then the declaration

```
P x;
```

does *not* mean that "x contains objects of type **P**," but rather that "x contains *pointers to* objects of type **P** (or null)." If **Q** is a subtype of **P**, furthermore, then the type "pointer to **Q**" is a subtype of "pointer to **P**." However, because it is extremely

burdensome always to be saying “*x* contains a pointer to *P*,” the universal practice is just to say “*x* is a *P*.” After this section, I’ll do that, too, but until it becomes automatic, I suggest that you consciously translate all such shorthand phrases into their full equivalents.

All this discussion should make it clear that the tag on a value can differ from the tag on a container that holds that value. This possibility causes endless confusion, because of the rather loose terminology that arose in the days before object-oriented programming (it is object-oriented programming that gives rise to cases where the confusion occurs). For example, the following Java program fragment introduces a variable (container) called *x*; says that the container’s type is (pointer to) *P*; and directs that a value of type (pointer to) *Q* be placed in *x*:

```
P x = new Q ();
```

Programmers are accustomed to speak of “the type of *x*.” But what does this mean: the type of the value contained in *x* (i.e., pointer to *Q*), or the type of the container itself (i.e., pointer to *P*)?

We will use the phrase “the *static type* of *x*” to mean the type of the container (in the example above, this type is “pointer to *P*”), and the phrase “the *dynamic type* of *x*” to mean the type of the value contained in *x* (in the example above, “pointer to *Q*”). This is an extremely important distinction! Object-oriented programming in Java or C++ will be a source of unending confusion to you until you understand it completely.

### 4.2.2 Type denotations in Java

A *type denotation* is a piece of syntax that names a type.

#### Syntax.

*Type*: *PrimitiveType* | *ReferenceType*

*PrimitiveType*:

**boolean**

**byte** | **char** | **short** | **int** | **long**

**float** | **double**

*ReferenceType*:

*ClassOrInterfaceType* | *ArrayType*

*ClassOrInterfaceType*: *Name* *TypeArguments*<sub>opt</sub>

*ClassType*: *Name* *TypeArguments*<sub>opt</sub>

*InterfaceType*: *Name* *TypeArguments*<sub>opt</sub>

*ArrayType*: *Type* *Dim*

*Dim*: [ ]

As the names suggest, the *Name* in a *ClassOrInterfaceType* must be the name of a class or interface, the name of a class in a *ClassType*, and of an interface in an *InterfaceType*. We’ve already seen examples of defining simple classes and interfaces. See §10 for the significance of *TypeArguments*.

---

**Cross references:** *TypeArguments* 234.

### 4.3 Environments

In order to direct a computer to manipulate something, you have to be able to mention that thing in your program. Programming languages therefore provide various ways to *denote* values (literal constants, such as 42 or 'Q') and to denote (or *name*) containers. Within our model, we can imagine that at any given time, there is a set of containers, which I will call the *current environment*, that allows the program to get at anything it is supposed to be able to reach. In Java (and in most other languages as well) the current environment cannot itself be named or manipulated directly by a program; it's just used whenever the program mentions the name of something that is supposed to be a container. The containers in this set are called *frames*. The named component containers inside them are what we usually call local variables, parameters, and so forth. You have already seen this concept in CS 61A, and might want to review the material from that course.

When we have to talk about environments, I'll just use the same container notation used in previous sections. Occasionally, I will make use of “free-floating” labeled containers, such as

X: 42

to indicate that X is a variable, but that it is not important to the discussion what frame it sits in.

### 4.4 Applying the model to Java

As modern languages in the Algol family go, Java is fairly simple<sup>3</sup>. Nevertheless, there is quite a bit to explain. Here is a summary of how Java looks, as described in the terminology of our model. We'll get into the details of what it all means in a later note.

- All simple containers contain either numeric values, booleans, or pointers (known as *references* in Java). (There are also functions, but the manipulation of function-valued containers is highly restricted, and not entirely accessible to programmers. We say no more about them here.)
- All simple containers are named and only simple containers are named. The names are either identifiers (for variables, parameters, or fields) or non-negative integers (for array elements).
- All simple containers have well-defined initial values: 0 for numerics, **false** for booleans, and **null** for pointers.
- The referents of pointers are always anonymous structured containers (called *objects* in Java).

---

<sup>3</sup>Algol 60 (ALGOrithmic Language) was the first widely used language with the kind free-format syntax familiar to C, C++, and Java users. It has, in fact, been called “a marked improvement on its successors.”

- Aside from environments, objects are created by means of the **new** expression, which returns a pointer (initially the only one) to a new object.
- Each container has a static type, restricting the values it may contain. A container's type may be *primitive*—which in Java terminology means that it may one of the numeric types or **boolean**—or it may be a *reference type*, meaning that it contains pointers to objects (including arrays). If a container's static type is primitive, it is the same as its dynamic type (that is, the type of the container equals the type of the value). If a container has a reference type, then its dynamic type is a subtype of the container's type.
- Named containers comprise local variables, parameters, instance variables, and class variables. Every function call creates a *subprogram frame*, (or *procedure frame*, or *call frame*), which contains parameters and local variables. The **new** operator creates *class objects* and *array objects*, which contain instance variables (also called *fields* in the case of class objects and *elements* in the case of arrays). The type of a class object is called, appropriately enough, a *class*. Each class has associated with it a frame that (for lack of a standard term) I will call a *class frame*, which contains the class variables (also called *static variables*) of the class.

## Chapter 5

# Declarations

*Declarations* introduce names into programs. Java has three kinds of declarations that introduce containers—field declarations (which introduce instance variables and class variables, §5.4.4), local variable declarations (§5.2), and formal parameter declarations (§5.8)—plus declarations for classes (§5.4), interfaces (§5.5), and various kinds of functions—methods (§5.8), constructors (§5.9), and static initializers (§5.10). With the exception of static initializers (which have an implicit name) all declarations associate *Identifiers* with some declared entity

### Syntax.

*Identifier*: *Letter LetterOrDigit\**

*Letter*: Unicode letter or `_`

*LetterOrDigit*: *Letter* | Unicode digit

The dollar sign (\$) also counts as a letter, but you should not use it in any program you write; it is intended to be used only in programs that are written by other programs. Upper-case letters are considered to be different from their lower-case counterparts, so the identifiers `x` and `X` are considered different. The Unicode letters and digits include not just the ASCII Latin letters and arabic digits, but also all letters and digits from other alphabets that are included in the Unicode standard. Of course, it is silly to use these other characters unless your program editor happens to display them properly.

Certain strings of letters are *reserved words* and may not be used as *Identifiers*. Here is a complete list<sup>1</sup>:

<code>abstract</code>	<code>default</code>	<code>goto</code>	<code>null</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>this</code>	<code>throw</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>package</code>	<code>throws</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>private</code>	<code>transient</code>
<code>case</code>	<code>extends</code>	<code>instanceof</code>	<code>public</code>	<code>true</code>

---

<sup>1</sup>The official Java specification divides this list into *keywords*, which each have unique grammatical purposes, and the literals `true`, `false`, `null`, which are like pre-defined identifiers. Two keywords—`goto` and `const`—are not actually used anywhere in Java at the moment, but are forbidden, presumably, to keep users out of trouble.

catch	false	int	return	try
char	final	interface	short	void
class	finally	long	static	volatile
const	float	native	super	while
continue	for	new	switch	

## 5.1 Scope and the Meaning of Names

All declarations have a *scope*: a range of program text in which they are in effect. First, the scope of a declaration has an immediate scope that is dictated by *block structure*. Second, the scope of individual declarations may be extended to arbitrary points in a program by *selection* and *field access*. Finally, the scope of declarations defined in classes and interfaces may be extended by *inheritance*. We’ll deal with field access in §6.6.2 and with inheritance in §5.4.2. After looking at block structure and selection, we put it all together to see how uses of names get their meanings in general.

### 5.1.1 Block structure

Within a single source file, most declarations in Java obey *block-structured scope rules*. The text can be thought of as being divided into nested *declarative regions*, and the scope of a declaration stretches to the end of that region starting either from the declaration itself or from the start of the declarative region.

To see what this all means, take a look at the code skeleton in Figure 5.1. Each declaration’s scope (on the left) is contained inside some declarative region (on the right), namely the *innermost* (smallest) region that contains the declaration. Declarations of classes, fields, functions, parameters, and statement labels have scopes that fill their entire declarative region. Declarations of local variables start at the point of declaration<sup>2</sup>.

The scopes of the field named `x` and the local variable named `x` overlap. In this case, the innermost (smaller) scope *shadows* (or *hides*) the outer one; we sometimes refer to the situation as a “hole in the scope” of the outer declaration. In general, whenever the overlap of two declarations creates an ambiguity that can’t be resolved from context, the one with innermost scope hides the other. Not all declarations with the same name hide each other, since the contexts in which some uses of names appear serve to distinguish which of several possible declarations apply. For example, there is never confusion between a field named `f` and a function named `f` because uses of method names must always be followed by argument lists. Also, two functions may have the same name and not be confused because they take different numbers or types of arguments (this is called *overloading* the function name, and is discussed further in §5.8.3).

---

<sup>2</sup>Yes, you could say that parameters and statement labels follow the same rule as local variable declarations since they happen to appear at the beginnings of their declarative regions. Take your pick.



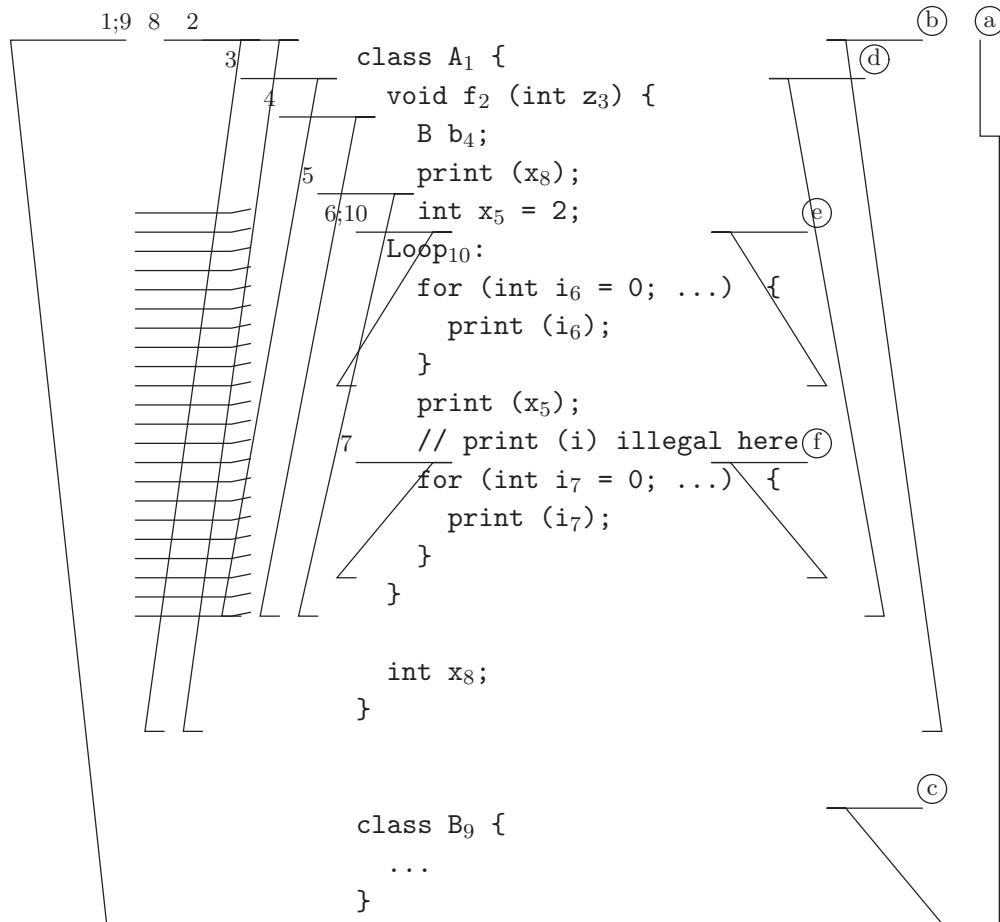


Figure 5.1: Block-structured scope rules in Java. The lettered brackets at the right show the declarative regions, and the numbered brackets at the left show the scopes of declarations. Region (a) is the entire file (file scope); (b) and (c) are class bodies; (d) is the body and parameters of function `f`; and (e) and (f) are the `for` loop bodies. The subscripts on identifiers link identifiers that are subject to the same declaration, and also refer to the scopes on the left. The cross-hatched portion of the scope of declaration #8 indicates where that declaration is *shadowed* by declaration #5 for the same identifier.

### 5.1.2 Selection

In addition to the block-structured scope rules illustrated in Figure 5.1, the scope of a declaration may extend to nearly any point in a program (even into another file) as the result of field access (discussed in §6.6.2) or of *selection* in a *qualified name*:

#### Syntax.

*Name*: *Qualifier*<sub>opt</sub> *SimpleName*

*Qualifier*: *Name* .

*SimpleName*: *Identifier*

In general, the syntax *A.I* in Java, where *I* is an identifier, means “the entity named *I* that is defined in the entity named or referenced by *A*.” When *A* is the *Name* in a *Qualifier*, as here, it may denote a class or interface, as in `java.lang.System.out`, where the qualifier `java.lang.System` denotes one of the standard library classes. It may also denote a package: both `java` and `java.lang` denote packages. (The other case, where it denotes a value, I am calling a field access (§6.6.2), rather than a qualified name.) The scope of declarations in the type or package denoted by the *Qualifier* is, in effect, extended to the point just after the dot (I suppose one might call it a “wormhole in scope.”)

Names, again, denote declarations of things. In many contexts, only names of particular kinds of things are allowed. We’ll denote these by using nonterminals with suggestive names:

*TypeName*: *Name* *TypeArguments*<sub>opt</sub>

*PackageName*: *Name*

*MemberName*: *Name* *TypeArguments*<sub>opt</sub>

*StaticMemberName*: *Name* *TypeArguments*<sub>opt</sub>

*MethodName*: *Name*

*ExpressionName*: *Name*

which you are to read as a name that denotes a type, package, member of a type, static member of a type, method, or variable, respectively. We’ll discuss *TypeArguments* more in Chapter 10.

There will be times when this (context-free) syntax alone is ambiguous. For now, For example, in `x.z = 3`, `z` must be an *ExpressionName*, but `x` could be either a *TypeName* or *ExpressionName*. Fortunately, the compiler has other ways to figure this out.

### 5.1.3 Packages and Imports

Besides the block structure that is evident in a single source file, Java imposes a kind of global block structure as well that defines how some names are visible in many source files. All types in Java are themselves contained in *packages*. A package is nothing more than a set of named types (classes and interfaces) and named

---

**Cross references:** *Identifier* 79.

**Cross references:** *TypeArguments* 234.

(sub)packages. No two entities in a package may have the same name. There is an outer-level anonymous package, which may be thought of as the root of a hierarchy of types and packages.

### Syntax.

*PackageClause*: *Annotation*\* **package** *PackageName* ;

We'll discuss *Annotations* separately in §5.13.

Packages are not declared explicitly. Instead, each source file defines which package its types are part of, using a *PackageClause* (see also §5.12 for its placement). Types from source files that don't specify a package go into the anonymous package. The types and packages in this anonymous package are what I'll call *outer-level* types and packages.

The precise initial contents of the anonymous package and its subpackages depends on the implementation of Java you are using. Typically, though, there are two ways in which such implementations represent the package hierarchy. In one, files represent classes and directories represent packages. One gives the compiler (or, during execution, the run-time system) a set of directories (a *class path*) whose contents represent the anonymous top-level directory. The files in that directory are the compiled versions of classes in the anonymous package or their source files; the subdirectories are the subpackages of the anonymous package. The definition is recursive. The other way to represent a class hierarchy is as a *class archive*—some kind of file that contains a set of classes, indexed by their names and packages (Sun calls these *jar files*: Java archives).

Certain packages form the large Java *standard library* of classes and interfaces. Covering the entire multitude is far beyond the scope of this book, but we will look at some of them in later chapters. Chief among these is the package `java.lang` (that is, “the package `lang` defined in the outer-level package `java`”). This package contains the types `String` (§9.2) and `Object`, plus all the exception types thrown implicitly by certain instructions in the language (see §8). In other words, `java.lang` includes all the classes needed to define the core of Java.

Subject to certain *access rules* (see §5.11), one can refer to any outer-level type or package by its name (a simple identifier) and to any other type or subpackage by selection (§5.1.2). Thus, you can write

```
java.lang.String msg = "Hello, world!";
java.io.Reader R = new java.io.FileReader (name);
```

This is entirely appropriate for types that are mentioned infrequently, but the reader of a program only needs to be reminded a few times where to find a particular type before the constant qualification becomes burdensome, so there are shortcuts:

- As we've seen, one can refer to any type defined in a particular source file from anywhere else in that file using just its simple name.

- One can refer to any type defined in a particular package from anywhere else in that package without qualification (that is, using just its simple name).
- Finally, one can refer to members of a package of class by their simple names with the help of **import** clauses.

Import clauses go at the beginning of a source file, just after any package declaration:

```
ImportClause: TypeImport | PackageImport | StaticImport
TypeImport: import PackageName . TypeName ;
PackageImport: import PackageName . * ;
StaticImport:
    import static PackageName . StaticMemberName ;
    import static PackageName . TypeName . * ;
```

These clauses simply allow you to refer to certain things by their simple names, without having to use selection. That's all they do: you don't need to import something in order to use it and importing does not give you extra access to anything. For example, the following four versions of `nextRoot` are equivalent:

```
// Version 1:
class Sqrt {
    /** The square root of the next number on INP. */
    double nextRoot (java.util.Scanner inp)
        return java.lang.Math.sqrt (inp.nextDouble ());
}
```

---

```
// Version 2:
class Sqrt {
    /** The square root of the next number on INP. */
    double nextRoot (java.util.Scanner inp)
        return Math.sqrt (inp.nextDouble ());
}
```

---

```
// Version 3:

import java.util.Scanner;
import static java.lang.Math.sqrt;
class Sqrt {
    /** The square root of the next number on INP. */
    double nextRoot (Scanner inp)
```

```

        return sqrt (inp.nextDouble ());
    }

```

---

```

// Version 4:

```

```

import java.util.*;
import static java.lang.Math.*;
class Sqrt {
    /** The square root of the next number on INP. */
    double nextRoot (Scanner inp)
        return sqrt (inp.nextDouble ());
}

```

In Version 1, I wrote out the full names of class `Scanner` and the static method `sqrt`. In Version 3, I used two **import** clauses to abbreviate the names `Scanner` and `sqrt`. In Version 4, I used `.*` to import *all* the public names<sup>3</sup> in the package `java.util`, and all the public static members in the class `java.lang.Math`. As Version 2 illustrates, finally, that every program implicitly starts with the *PackageImport* clause `import java.lang.*`.

Import clauses do not import packages. In fact, you can never refer to subpackages by their simple names, except for outer-level packages.

These rules as stated lead to certain ambiguities: for example, what if a type is introduced by a *TypeImport* and has the same name as a type in the same package? In general, names defined in the same source file take precedence over names defined by the non-asterisked **import** clauses, which take precedence over type names from the same package, which take precedence over the asterisked **import** clauses.

## 5.2 Local Variables

A *local variable* is a simple container whose scope is delimited by some block and which goes out of existence when execution of its block is finished.

### Syntax.

*LocalDeclarationStatement:*

**final**<sub>opt</sub> *Type* *InitDeclarator*<sup>+</sup> ;

*InitDeclarator:* *Declarator* *VarInit*<sub>opt</sub>

*Declarator:* *SimpleName*

*VarInit:*

= *Expression*

= *ArrayInitializer*

---

<sup>3</sup>See §5.11 for the meaning of “public.”

The scope of a local declaration starts at the point of declaration and goes to the end of the innermost enclosing declarative region (usually to the curly brace that ends whatever block it is in). No *SimpleName* used in a *Declarator* may duplicate that of any other local declaration or parameter that is in scope, even one in an enclosing block. Thus, local declarations may not hide each other and may not hide parameter declarations. Local declarations may hide field declarations (§5.4.4) and (where there is ambiguity) type declarations<sup>4</sup>. Statement labels (§7.5) and method declarations never create ambiguities with local variables, so duplication of names here is perfectly legal (even if usually unwise).

The effect of the declaration is to create new containers, one for each *Declarator*, and to initialize their contents to the value given in their respective *VarInits*, where present. The *Type* part of a declaration gives the static type of the new local variables named by the *Declarators*<sup>5</sup>.

A *VarInit* provides an initial value for a local variable at the point of declaration. The *Expression* is evaluated and its value assigned, just as for an ordinary assignment (see §6.7). The *ArrayInitializer* option—as in

```
int[] x = { 1, 2, 3 };
```

—is a shorthand. The declaration above, for example, is equivalent to

```
int[] x = new int[] { 1, 2, 3 };
```

(see §6.6.4) and allows you to create and initialize a new array in one statement.

A local variable must be assigned a value before its value is used, and it must be apparent to the compiler that this is so. See §6.11 for what “apparent to the compiler” officially means.

The inclusion of the modifier **final** in a local-variable declaration indicates that the variable will not be changed once it is explicitly initialized (either by a *VarInit* or later assignment). The compiler will complain if it cannot be sure that there will be only one assignment to such a variable.

**Stylistic matters.** I suggest that you use initialized variables only in the *ForInit* clauses (§7.4) in **for** loops and for **final** local variables or variables you do not intend to change once set. This is a matter of taste, but to my eye, inclusion of a value in a declaration such as

```
int N = 0;
```

suggests that N is being defined to be 0.

---

**Cross references:** *Type* 76, *Expression* 119, *ArrayInitializer* 150, *SimpleName* 82.

<sup>4</sup>The hiding of type declarations by local variables is one of those really obscure cases that really shouldn’t come up, since the usual conventions for naming classes and local variables keeps their names different.

<sup>5</sup>In fact, there is alternative syntax, held over from C, for local variables that denote arrays. The declaration `int x[]` is synonymous with `int[] x`. As a stylistic matter, however, I ask that you not use this older form, and so have not included it in the syntax.

## 5.3 Type Declarations

The set of Java types is divided into primitive types (**int**, etc.) and reference types. The language provides a built-in way to create one kind of reference types: array types are introduced automatically whenever their types are mentioned. The programmer can define the other kinds of reference types, class and interface types, using class and interface declarations.

### Syntax

*TypeDeclaration*: *ClassDeclaration* | *EnumDeclaration* | *InterfaceDeclaration*

## 5.4 Class Declarations

A class declaration introduces a new type of structured container.

### Syntax.

*ClassDeclaration*: *ClassModifier*\* *BasicClassDeclaration*

*BasicClassDeclaration*:

**class** *SimpleName* *TypeParameters*<sub>opt</sub>  
           *Extends*<sub>opt</sub> *Implements*<sub>opt</sub>  
           *ClassBody*

*ClassModifier*: **final** | **public** | **abstract** | *Annotation*

*Extends*: **extends** *ClassType*

*Implements*: **implements** *InterfaceType*<sup>+</sup>,

*ClassBody*: { *ClassBodyDeclaration*\* }

*ClassBodyDeclaration*:

*FieldDeclaration*  
*MethodDeclaration*  
*ConstructorDeclaration*  
*MemberTypeDeclaration*  
*StaticInitializer*  
*InstanceInitializer*

(We'll discuss *TypeParameters* in Chapter 10.) This syntax defines a kind of template for *instances* of the class (created by the **new** operator, §6.6.1). Each instance is a structured container whose subcontainers are the same for all instances of the class and are given by its field declarations. In addition, the class declaration specifies a set of functions (called *methods*) that are applicable to instances of the type, a set of *constructors*, which are functions that initialize (set the fields of) new instances of the the type, and static initializers for initializing *class variables* shared

---

**Cross references:** *ClassDeclaration* 87, *InterfaceDeclaration* 91, *EnumDeclaration* 92.

**Cross references:** *Annotation* 116, *ClassType* 76, *ConstructorDeclaration* 107, *FieldDeclaration* 90, *InterfaceType* 76, *MemberTypeDeclaration* 94, *MethodDeclaration* 99, *StaticInitializer* 112, *SimpleName* 82, *TypeParameters* 234.

by all instances. Finally, one can define classes inside the class; this feature is intended for classes that are needed to implement the outer class but would unduly clutter up the package if they were written as full-fledged classes.

### 5.4.1 Kinds of class members

Collectively, the fields, methods, constructors, and member classes defined in a class are known as its *members*. There are two kinds of members: *static* and *non-static*.

Static members (indicated with the keyword **static**) are associated with the type itself; they are one-of-a-kind. Thus, static fields are also called *class variables*, and static methods are called *class methods*. Non-static members (indicated by the absence of the keyword **static**) are associated with individual *instances* of the type. Thus, non-static fields are also called *instance variables* and non-static methods are also called *instance methods*. To see the difference, consider an accumulator class:

```
class Accum {
    static int total = 0;
    int sum = 0;
    static void resetTotal () {
        total = 0;
        // Illegal to mention sum here.
    }
    int incr (int n) {
        sum += n; total += n;
    }
}
```

and the following program:

```
Accum a1 = new Accum ();
a1.incr (3);
// Now a1.sum, a1.total, and Accum.total = 3.
// Accum.sum is illegal.
Accum a2 = new Accum ();
a2.incr (2);
// Now a1.sum = 3, a2.sum = 2, and
// a1.total, a2.total, and Accum.total = 5.
a1.resetTotal (); // or a2.resetTotal()
                  // or Accum.resetTotal()
// Now a1.sum = 3, a2.sum = 2, and
// a1.total, a2.total, and Accum.total = 0.
```

The three variables `a1.total`, `a2.total`, and `Accum.total` are all the same, single variable; there is only one for the class `Accum`. The variables `a1.sum` and `a2.sum` are distinct; there is one for every instance created by `new Accum()`. It is illegal to refer to `Accum.sum` because it makes no sense: the `sum` variable of *which* instance of `Accum`? The non-static member function `incr` increments the `sum` variable of a



particular instance. The static member function `resetTotal` can touch only the static variables of its class.

### 5.4.2 Inheritance

When a class *C* extends another class *P*, We say that *P* is the *direct superclass* of *C*, and *C* is a *direct subclass* of *P*. A class without an **extends** clause automatically has `Object` (in package `java.lang`) as its direct superclass. The set of all superclasses of *C* consists of *C*, *P*, and all superclasses of *P* recursively; we use the term *proper superclass* if we want to exclude *C* itself. Similarly for subclasses and proper subclasses. Thus, only the class `Object` has no proper superclasses, and all classes are subclasses of `Object`.

If *P* is a superclass of *C*, we also say that *C* is a *subtype* of *P*. If *P* is not *C* itself, then *C* is a *proper subtype* of *P*.

The main importance of the superclass relationship is that it controls *inheritance*. The fields of class *C* consist of all the fields of *P* (its superclass) plus any that are explicitly declared in *C* itself. We say that *C inherits* the fields of *P*. Thus, each instance of *C* will contain at least the same instance variables as a *P*, and *C* will share *P*'s class variables (that is, if *P.x* is a class variable of *P*, then *C.x* will refer to the same variable)<sup>6</sup>.

Likewise, *C* inherits the non-private methods of *P*. Here, however, there is an additional, extremely important wrinkle. If a non-static method declaration in *C* has the same name and argument types as one that is inherited from *P*, then we say that *C overrides* the definition of that method. See §5.8.4 for the meaning of this.

The class *C* also inherits the fields and methods from the interfaces that it implements, and can override methods from these interfaces; see §5.5.

### 5.4.3 Class Modifiers

The class modifiers specify certain properties of the definition. The modifiers **final** and **abstract** are mutually exclusive. We'll discuss *Annotations* later in §5.13.

A *public* class may be mentioned directly in compilation units (source files) for other packages. Non-public classes may not. See §5.11 for a complete discussion of the **public** keyword and other *access modifiers*. The general convention is that a given compilation unit may contain at most one public class.

A class designated to be **final** may not be extended. That is, it may not appear in the **extends** clause of any other class. In the abstract, there is seldom any particular reason to do this, closing off forever the possibility of extending a certain class. However, it does allow some implementations of Java to make some operations on the class a little faster.

An *abstract class* may not have instances. That is, it is illegal to execute `new C(...)` for any abstract class *C*. On the other hand, the array allocator **new**

---

<sup>6</sup>If one declares a field in *C* with the same name as one in *P*, then *C* will contain two distinct fields with the same name. The rules of Java will tell you what this means, but it is very confusing, so don't ever do it.

$C[\dots]$  is perfectly legal, since it does not create any  $C$ 's (see §6.6.4). The idea behind an abstract class is that it represents a *common interface* that is shared by all its subclasses, which extend it to fill in concrete behavior. Accordingly, not all the methods in an abstract class need to have bodies. A variable's type may be an abstract class, even though values stored in that variable cannot point to an object of exactly that type (since there aren't any); by the rules of Java, however, that variable *can* contain values of the subclasses.

#### 5.4.4 Fields

##### Syntax.

*FieldDeclaration:*

*FieldModifier*\* *Type* *InitDeclarator*<sup>+</sup> ;

*FieldModifier:*

**public** | **protected** | **private**

**final**

**static**

**volatile** | **transient**

*Annotation*

Fields are containers (see §4.1.1); that is, they are a form of variable and contain values. You may either have a single instance of a field for the entire program (**static** fields (or *class variables*)) or one instance of a field for every object of the class (non-static fields or *instance variables*), as described in §5.4.1. Class variables are created and initialized when the class itself is loaded (see §5.10), and instance variables are created each time the **new** operator creates a new object. If a field declaration contains a *VarInit* (see §5.2), then that provides the initial value for the field, as described in more detail in §5.9. Otherwise, when the field is created, it is given an initial value that depends on the type: 0 for numeric types, **false** for **booleans**, and **null** for reference types.

A **final** field may be assigned to exactly once, after which further assignments are illegal (so that the field's value remains constant). Typically, one initializes **final** fields with a *VarInit*, but the current definition of Java allows you to initialize them in a constructor, as long as there is only one assignment.

The keywords **private**, **protected**, and **public** control where the programmer may refer to a field. We discuss these in §5.11.

The **volatile** keyword indicates that from the point of view of any given thread (see Chapter 11) the variable can change at any time. You will seldom have to worry about this, fortunately; see §11.3.3 for more discussion.

The **transient** keyword is even more obscure. It means that the value of the field would be meaningless outside this particular execution of this particular program, so there is generally no point in trying to write it to a file, for example, to be read back later. No doubt you are burning with curiosity to know when you would ever use it, but I'm afraid we are going to disappoint you.

## 5.5 Interface Declarations

An *interface* is similar to an abstract class in that it is impossible to create an object whose type is an interface type. It differs in that *all* methods in an interface are abstract (and must have no bodies), and all fields are static constants.

### Syntax.

*InterfaceDeclaration*: *InterfaceModifier*<sup>\*</sup> *BasicInterfaceDeclaration*

*BasicInterfaceDeclaration*:

**interface** *SimpleName* *TypeParameters*<sub>opt</sub>

*SuperInterfaces*<sub>opt</sub>

{ *InterfaceBodyDeclaration*<sup>\*</sup> }

*InterfaceModifier*: **public** | **abstract** | *Annotation*

*SuperInterfaces*: **extends** *InterfaceType*<sup>+</sup>;

*InterfaceBodyDeclaration*:

*ConstantDeclaration*

*MethodSignatureDeclaration*

*MemberTypeDeclaration*

*MethodSignatureDeclaration*:

*InterfaceModifier*<sup>\*</sup> *MethodSignature* ;

*ConstantDeclaration*: *ConstantModifier*<sup>\*</sup> *Type* *InitDeclarator*<sup>+</sup> ;

*ConstantModifier*: **public** | **static** | **final** | *Annotation*

In a *ConstantDeclaration*, all *InitDeclarators* must have a *VarInit* clause (=...). Since interfaces are always abstract, the **abstract** modifier is superfluous, and as a matter of style should not be used. All *ConstantDeclarations* in an interface declaration are public, static, and final, so the keywords **public**, **static**, and **final** are superfluous, and should not be used. Finally, the **public** and **abstract** keywords applied to methods are superfluous (all interface methods are public and abstract) and should not be used<sup>7</sup>.

An interface can extend any number of other interfaces, which are called its *direct superinterfaces*<sup>8</sup>. An interface's complete set of superinterfaces consists of itself, its direct superinterfaces, and all of their superinterfaces (recursively). An interface is a subtype of its superinterfaces. As for classes, the significance of superinterfaces is inheritance: an interface inherits all methods and fields declared in its superinterfaces. Methods having the same name, argument types, and return type and defined in two different superinterfaces are considered to be a single method (this makes sense only because methods in interfaces never have bodies.)

Interfaces exist to be *implemented* by classes. The **implements** list in a class

---

**Cross references:** *Annotation* 116, *InitDeclarator* 85, *InterfaceType* 76, *MemberTypeDeclaration* 94, *MethodSignature* 99, *SimpleName* 82, *TypeParameters* 234.

<sup>7</sup>I know that's a lot of superfluous syntax that you are never supposed to use; it's the usual thing—"historical reasons."

<sup>8</sup>WARNING: This terminology is really confusing in one sense. An interface contains, in general, *more* members than its *superinterface*. The term makes sense because the set of classes that implement a superinterface of interface *I* is a superset of the set of classes that implement *I*.

declaration specifies a set of interfaces the class implements (and inherits from). The class also implements all superinterfaces of these interfaces as well. A class is a subtype of all interfaces it or one of its superclasses implements.

## 5.6 Enumerated Types

An *enumerated* (usually abbreviated *enum*) class is a type of class whose main content is a set of distinct named constants of that class. These types are intended for situations where one wants a type whose purpose is to represent a relatively small set of values whose main property is simply that they differ from each other.

### Syntax.

```
EnumDeclaration: ClassModifier* enum SimpleName
                  Implementsopt {
                      EnumConstant*, EnumBodyDeclarationsopt }
EnumBodyDeclarations: ; ClassBodyDeclaration*
EnumConstant: SimpleName Formalsopt ClassBodyopt
```

Enum classes may not be abstract. When nested (see §5.7), they are always static.

**Semantics** First, let's consider an example of the simplest case:

```
package Rainbow;

enum Color {
    Red, Yellow, Green, Blue, Violet
}
```

This defines a class `Color` and five static final (constant) members that are automatically initialized to be different values of type `Color`. You can refer to them in the usual fashion for static members: as `Color.Red`, etc. However, you will probably find it easier to include a static import at the beginning of your program:

```
import static Rainbow.Color.*;
```

so that you can simply write `Red`, `Blue`, etc.

This simple declaration provides you with the usual methods of `Object`. Since the values of the enumeration constants (or *enumerals*, as they are sometimes called) are all distinct, we have that `Red != Blue`, and so forth<sup>9</sup>. Finally, there are few methods that are peculiar to enumerated types:

`Color.values ()` is an array containing the values `Red` through `Blue` in the order they are listed.

---

**Cross references:** *ClassModifier* 87, *Implements* 87, *ClassBody* 87, *ClassBodyDeclaration* 87, *Formals* 100

<sup>9</sup>For some reason, some beginning programmers are tempted to compare values of enum types in a convoluted fashion. For example, instead of `x==Red`, one might see `x.ordinal() == Red.ordinal()` or even `x.name().equals("Red")`. Don't do that.

**v.ordinal ()** where **v** is a **Color**, is the position of **v** in the array **Color.values ()** (so **Red.ordinal ()** is 0).

**v.name ()** is the declared name (a **String**) for the **Color** value **v**.

**Colors.valueOf (s)** is the **Color** value whose enumeral is named **s** (a **String**). Thus, **s.equals(Colors.valueOf(s).name())**, assuming there is such an enumeral.

By special dispensation, Java allows you to use enumerals in **switch** statements:

```
switch (someColor) {
case Red:
    return 751;
case Yellow:
    return 587;
case Green:
    return 534;
case Blue:
    return 473;
case Violet:
    return 422;
}
```

(which might be a way to translate **Colors** to approximate wavelengths in nanometers).

Most of this so far looks like a simple elaboration of enumerated types in other languages, such as C, Pascal, or Ada. However, Java goes further, in that enumerated types are full-fledged classes, and can contain arbitrary data besides their names and ordinal positions. The easiest elaboration is the addition of instance variables and constructors. For example, suppose **Colors** carry around the wavelengths of light they denote. Here's one way to do so:

```
public enum Color {
    Red (780, 722), Yellow (597, 577), Green (577, 492),
    Blue (492, 455), Violet (455, 390);

    /** A Color roughly corresponding to wavelengths between LONGEST
        and SHORTEST nanometers. */
    Color (int longest, int shortest) {
        this.longest = longest;
        this.shortest = shortest;
    }

    /** A representative wavelength in nanometers for light of
        this Color. */
    public int wavelength () {
```

```

        return (this.longest + this.shortest) / 2;
    }

    final int longest, shortest;
}

```

The `wavelength` method replaces the previous `switch` statement.

Though it's uncommon, we can go even further and supply enumerals with class bodies, which can, as usual override methods of the `Color` class. The effect is to extend the class `Color` with anonymous types, so that the enumerals don't all necessarily have the same dynamic types, although their types are all subtypes of `Color`.

## 5.7 Nested Classes and Interfaces

Class and interface declarations may be nested inside other constructs; we call them *nested classes* and *nested interfaces* to distinguish them from *top-level* classes and interfaces. Such declarations may be members of classes and interfaces, called *member classes* and *member interfaces*. Nested classes may be defined in blocks, in the same positions as local variable declarations, in which case they are *local classes*. Member classes and interfaces may be declared **static**, which means essentially that they are ordinary type declarations that can be named by selection like other static class members and that have access to **private** fields of the class that contains them. Nested classes that are non-static are called *inner classes*; they also have access to instance (non-static) variables of their enclosing definition.

### 5.7.1 Member types

**Syntax.**

*MemberTypeDeclaration:*

*MemberTypeModifier*\* *BasicClassDeclaration*

*MemberTypeModifier*\* *BasicInterfaceDeclaration*

*MemberTypeModifier:*

**private** | **protected** | **public**

**static** | *Annotation*

Member interfaces are implicitly static; the use of the **static** keyword for them is allowed. At most one access modifier (**private**, **protected**, **public**) is allowed. Member classes that are defined inside interfaces are implicitly **public** and **static**; again these modifiers are allowed, but unnecessary.

There are restrictions on the contents of member types. Inner classes may not declare static initializers (§5.10) or static member types. They may not declare static fields, unless they are declarations of constants (§6.9).

**Semantics.** A static member type is essentially the same as a top-level type, with a fancier name and a little more access.

```
class Outer {
  private static int x;
  private int y;
  static class Nested {
    static int z = x;      // OK
    // static int r = y;   // ILLEGAL
    void f (Outer a) {
      a.y = 3;             // OK
    }
  }
}
```

The full name of the nested class here is `Outer.Nested`. The simple name `Nested` is valid only inside the declaration of `Outer`. The class `Nested` has access to all the static members (fields, functions, and other nested classes) of `Outer` by their simple names, but it can only access instance variables and other non-static members using selection or field access, (e.g., `a.y` in the example above).

An inner member class (non-static nested class) is a much different beast. Intuitively, the idea is supposed to be that each instance of the enclosing class has its own version of the inner class. To show all the mechanisms involved, it's best to look at an example. Figure 5.2 (page 96) shows a set of declarations for the top-level class `House` and the inner class `House.Door`, and a diagram of the result of executing `h = Builder.newHouse ("T. C. Mits")`. To create an inner class object, one uses a class allocator (§6.6.1) of the form

*OuterObject.new InnerObjectClass(...)*

as illustrated in the **new** expressions in the figure that create `Doors`. A pointer to the instance of the outer class (in this case, a `House`) with which inner instances (in this case, a couple of `Doors`) were created is stored in those inner instances, available as if it were a field named *OuterObjectType.this* (`House.this` in the example from the figure). Thus, instance methods of a `Door` can access the fields of the `House` it was created from. The usual block-structured scope rules apply, so that, as indicated in a comment, the methods of the inner class can refer to the fields of the outer class without using `House.this`.

### 5.7.2 Local classes

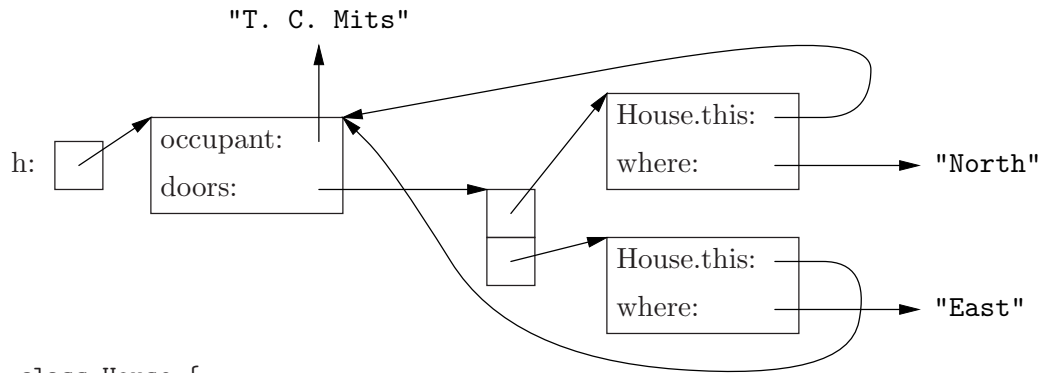
A *local class* is a class defined in a statement sequence, such as a block.

**Syntax.**

*LocalClassDeclaration: BasicClassDeclaration*

---

**Cross references:** *BasicClassDeclaration* 87.



```

class House {
    House (String occupant, int numDoors) {
        this.occupant = occupant;
        doors = this.new Door[numDoors];
        // Just 'doors = new Door[...]' would also be legal.
    }
    String occupant;
    Door[] doors;

    public class Door {
        Door (String where) { this.where = where; }
        String where;
        void enter () {
            System.out.println
                ("Entered house belonging to "
                 + House.this.occupant + " via the "
                 + this.where + " door.");
            // It would also be legal to have written
            // + occupant + " via the " + where + " door.");
        }
    }
    void addDoor(String where, int which) {
        doors[which] = this.new Door (where);
    }
}

class Builder {
    static House newHouse (String who) {
        House result = new House (who, 2);
        result.addDoor ("North", 0);
        // or I could have used
        // doors[0] = result.new Door ("North");
        result.addDoor ("East", 1);
        return result;
    }
    void checkOutHouse (House house) {
        house.doors[0].enter ();
    }
}

```

Figure 5.2: Result of `h = Builder.newHouse ("T. C. Mits")`



**Semantics.** A local class is always an inner class. The scope of its declaration is limited as for local variable declarations, so it cannot be named outside the statement sequence that contains it. Otherwise, local classes are like other inner classes, with one additional capability: they have access to final (constant) local variables and (within methods) to **final** parameters of the enclosing construct. For example,

```
void f (int x, final int y) {
    int z = 19;
    final int q = 42;
    class Inner {
        int a, b;
        void g () {
            a = q + y; // Legal
            // a = x;   // ILLEGAL
            // a = z;   // ILLEGAL
        }
    }
    ...
}
```

### 5.7.3 Anonymous classes

An *anonymous class* is essentially a local class (§5.7.2) that is used only once, and so does not need a name. It makes its appearance only in allocator expressions (see §6.6.1).

#### Syntax.

*AnonymousClassAllocator:*  
 $OuterObject_{opt} \text{ new } ClassOrInterfaceType ( Expression^*, )$   
 $Extends_{opt} Implements_{opt}$   
 $\{ ClassBodyDeclaration^* \}$   
*OuterObject:* *Operand* .

Anonymous class declarations may not explicitly define constructors, and must also conform to the usual restrictions on inner and local classes.

#### Semantics. Writing

`new T( $E_1, \dots, E_n$ ) { ... }`

is essentially like writing

`new ANONYMOUS( $E_1, \dots, E_n$ )`

after first having written the local declaration

---

**Cross references:** *ClassBodyDeclaration* 87, *ClassOrInterfaceType* 76, *Expression* 119, *Extends* 87, *Implements* 87, *Operand* 119.

```

class ANONYMOUS
    extends T
    // or implements T if T is an interface
{
    ANONYMOUS (T1x1, ..., Tnxn) {
        super (x1, ..., xn);
    }
    ...
}

```

It simply allows you to avoid some writing and to avoid creating a new name.

#### 5.7.4 What on earth is this all for?

I commend to you the exercise of figuring out how to get the essential effects of nested classes in their absence. For the most part, one can easily come up with top-level classes that do the same things. One might reasonably wonder, therefore, why anyone would bother with the rather substantial added semantics.

Static nested classes clearly serve only to avoid cluttering packages with names of classes that are used only in the implementation of a single class. They differ from top-level classes only in the fact that they can get at the private variables of the class they are nested in.

Inner member classes add a little more: they arrange automatically for special fields that contain pointers back to some “parent object.” Still, this does not seem to be an enormous service.

The really interesting cases are local classes and anonymous classes, because these do something that isn’t easily attained by other means in Java: give methods access to local variables of another method. They provide us with a kind of restricted *function closure*, such as is available in the Scheme language.

Suppose you wanted to write a program to select all elements of a list that meet certain criteria:

```

/** Return list of all x in A for which P.test(x) is true. */
List select (List A, Predicate p) {
    List result = new List ();
    List last;
    last = result;
    for (List L = A; L != null; L = L.next) {
        if (P.test (L.head))
            last = last.next = new List (L.head, null);
    }
    return result.next;
}

```

where `Predicate` is an interface:

```

public interface Predicate {

```

```

    /** True iff OBJ satisfies this Predicate. */
    boolean test (Object obj);
}

```

I can use these definitions to write a function that removes all instances of a given string from a list of strings:

```

/** Return result of removing all instances of X from L. */
public List removeAll (final String x, List L) {
    return select
        (L,
         new Predicate() {
             boolean test (Object obj) {
                 return ! obj.equals (x);
             }
         }
        );
}

```

Admittedly, this is not as concise as the Scheme equivalent:

```

(define removeAll (x L)
  (select L (lambda (obj) (not (equal? obj x)))))

```

but then, nothing ever is.

In Java programs, inner classes, including anonymous classes, see most of their use in providing *call backs* for GUI interfaces. The term “call back” refers to functions (or function-like things) that the programmer passes to the routines in a graphical user interface library to inform the system of actions to take whenever certain events occur.

## 5.8 Method (Function) Declarations

For reasons that I confess not to understand, where other programming languages use terms like “procedure,” “subprogram,” or “function,” object-oriented languages (including Java) use the term *method*. To avoid confusion, I’ll do so here, but bear in mind that there really isn’t anything new about methods *per se*. We’ll discuss the details of calling methods in §6.8. This section describes how to define them.

### Syntax

*MethodDeclaration:*

```

TypeParametersopt MethodModifier+ MethodSignature ;
TypeParametersopt MethodModifier* MethodSignature MethodBody

```

*MethodSignature:*

```

Type SimpleName Formals Throwsopt
void SimpleName Formals Throwsopt

```

*MethodModifier:*

```
public | protected | private
final | static | abstract
native
synchronized
Annotation
```

*Formals:*

```
( Parameter* )
( Parameter+, VarArgs )
( VarArgs )
```

*Parameter:* **final**<sub>opt</sub> *Type SimpleName*

*VarArgs:* **final**<sub>opt</sub> *Type ... SimpleName*

*Throws:* **throws** *ClassType*<sup>+</sup>,

*MethodBody:* *Block*

(We'll discuss *TypeParameters* in Chapter 10.) A method without a *MethodBody* must be either **abstract** or **native**. Abstract methods may not be **private**, **final**, **static**, **native**, or **synchronized**.

### 5.8.1 Variable-Length Parameter Lists

The trailing formal parameter is allowed to stand for an arbitrary number (zero or more) of actual parameters. A signature such as

```
public void printf (String format, Object... args)
```

is essentially the same as

```
public void printf (String format, Object[] args)
```

That is, in the body of **printf**, you can refer to the array **args** as usual. However, when you call this method, you have the option of providing, instead of a formal parameter of type **Object[]**, a sequence of 0 or more values compatible with type **Object**. The call

```
printf ("%s = %d%n", name, value);
```

is then equivalent to

```
printf ("%s = %d%n", new Object[] { name, value });
```

So, in effect, **printf** takes a variable number—one or more—of parameters. Using the terminology of the C language, **args** is a *varargs parameter*. The Scheme and Common Lisp languages have a similar feature.

Let's take a full example with an actual implementation:

---

**Cross references:** *Annotation* 116, *Block* 168, *ClassType* 76, *SimpleName* 82, *Type* 76, *TypeParameters* 234.

```

/** A String consisting of the concatenation of the items in ARGS,
 *  separated by SEPARATOR. */
public static String makeList (String separator, String ... args) {
    String result;
    result = "";
    for (int i = 0; i < args.length; i += 1) {
        if (i > 0)
            result += separator;
        result += args[i];
    }
    return result;
}

```

Now with this definition, we can write:

```

makeList (" ", "The", "quick", "brown", "fox")
    produces "The quick brown fox"

```

Also,

```

String[] words = { "The", "quick", "brown", "fox" };
makeList (" ", words) produces "The quick brown fox"

```

has the same result.

### 5.8.2 Method signatures

The *MethodSignature* part of a method declaration gives the *syntactic specification* of the method: how it is called and what kind of results it might have—everything except how it works. A method call (§6.8) looks like

$$f(E_1, \dots E_n)$$

or

$$X.f(E_1, \dots E_n)$$

In the applicable method signature,  $f$  is the *SimpleName*, and there are  $n$  *Parameters*, whose types must be compatible with the types of the expressions  $E_i$  (again, see §6.8). The *Type* tells you what type of value it returns<sup>10</sup>, according to the same rules as for local declarations described in §5.2 (the same rules also apply to the types of *Parameters*). The special “return type” **void** marks a function that does not return a value.

If the method has a *MethodBody*, then the *SimpleNames* in the *Parameters* give names for local variables that contain the parameter values, which then may be used in the body. These local variables are called the *formal parameters* of the

---

<sup>10</sup>There is an alternative syntax for returning arrays, inherited from C. One may write, for example, `int foo() [] []` instead of `int [] [] foo()`. I haven’t shown it because as a stylistic matter, I believe you shouldn’t use it. Similarly, I have left out the old syntax for array-valued parameters, e.g., `int A[]` (in fact, Dennis Ritchie himself regretted adding that syntax to C).

method. They may be assigned to like ordinary local variables, unless the parameter is marked **final**.

As described in Chapter 8, methods may, instead of returning a value, raise an exception. The *Throws* clause lists exceptions (or some of them) that may be thrown as a result of calling the method. See §8.3 for a fuller description.

**Method Modifiers.** See §5.11 for the meanings of **public**, **private**, and **protected**. A method that is **final** may not be overridden in any subclasses of a class. A method that is **abstract** has no body, and *must* be overridden in any non-abstract subclass (that is, in any class that can have instances). Only abstract classes may contain abstract methods. Classes can obtain methods by inheritance from superclasses and interfaces, so in non-abstract classes, such inherited abstract methods also have to be overridden.

Static methods (also called *class methods*) are like ordinary functions or procedures in other programming languages. They are not subject to overriding (§5.8.4). A static method **g** defined in a class **Y** like this:

```
static double g (double x) { ... }
```

is typically called like this:

```
Y.g(4.2)
```

where **Y** is a class containing the definition of static method **g** (possibly by inheritance).

Non-static methods (called *instance methods*) have, in effect, an extra, implicit parameter. An instance method **f** defined in a class **Y** like this:

```
double f (double x) { ... }
```

is typically called like this:

```
E.f(4.2)
```

where **E** yields a value of type **Y** (or some subclass) . Roughly it is as if **f** had been declared

```
static double f (final Y this, double x) { ... }
```

and called like this:

```
Y.f (E, 4.2)
```

(the special variable name **this** is reserved for this purpose). However, there is one major difference not captured by this approximate rewriting: the effect of overriding, covered in §5.8.4.

There are two ways to implement a method: one can supply a *MethodBody*, or one can declare the method to be **native**. A native method is what is sometimes called a *foreign function* in other languages. In effect, it is an extension of the Java interpreter or compiler you are using. It is typically implemented in some language

other than Java (such as C or assembly language) and can do things that no non-native Java method can do. How you actually supply the body of a native method depends on the implementation of Java you are using, and we will not cover them further in this document.

### 5.8.3 Overloading and hiding methods

It is possible to have multiple methods in the same class (explicitly or through inheritance) that have the same name. When two such methods have different numbers or types of argument types, we say that they *overload* each other. They are two entirely unrelated methods, and which one is referred to by a particular call depends on which best fits the arguments to that call. For example,

```
void f () { ... }           /* (1) */
int  f (String y) { ... }   /* (2) */
double f (int y) { ... }    /* (3) */
void f (char y) { ... }     /* (4) */
void f (int y, int y) { ... } /* (5) */
void f (int y, char c) { ... } /* (6) */
void f (char c, int y) { ... } /* (7) */
```

(these are instance methods, but the same rules apply to static methods and to constructors (§5.9). Consider the call

```
x.f()
```

(assuming *x* is a variable of the appropriate type). Just looking at *x.f*, it is ambiguous which *f* we are referring to. However, only declaration (1) specifies an *f* without parameters, so that is the method that is called. Likewise,

```
x.f("Hello")
```

refers unambiguously to declaration (2). In general, to figure out which method to call, Java will only consider methods in which the static types of the actual parameters (that is, the parameters in the call statement) are *call convertible* to the types of the formal parameters: that is, only if all possible values of the actual parameter type may be assigned to a variable having the formal parameter's type (see §6.2.2).

This rule still leaves some ambiguities to resolve. Since any **char** value can be contained in an **int**, the rule leaves two choices, (3) and (4), for the call

```
x.f ('c')
```

This ambiguity is resolved in favor of (4) by the second rule of overload resolution: given a choice between two methods, pick the *more specific* one: the one whose formal parameters' types are all call convertible to the formal parameter types of the other. Since every **char** value fits in an **int**, but not every **int** fits in a **char**, method (4) is more specific ("pickier").

Not all overloaded calls can be resolved. For example,

```
x.f ('a', 'b')
```

This call fits either (5), (6), or (7). However, while (6) is more specific than (5), and (7) is more specific than (5), neither (6) nor (7) is more specific than the other, so the call is ambiguous (and therefore erroneous).

Do these rules seem complicated? You wouldn't be alone in thinking so (although the corresponding rules in C++ make these look positively trivial). Some authorities go so far as to shun overloading altogether. I don't, but do by all means avoid obscure cases like the combination of (5)–(7).

Resolving which overloaded function to call is a static process. The actual values of the variables in a call are irrelevant. So, for example, if `a` is an `int` variable, then `x.f(a)` always refers to declaration (3), not (4), even if `a` happens to contain a value, like 3, that fits into a `char`.

We've dealt so far only with methods whose parameter lists differ. No two methods defined in the same class or interface may have the same number and types of parameters. If a class explicitly defines a static method with the same number and types and parameters as an inherited method, we say that the explicit definition *hides* the inherited one. Calls are unambiguously resolved to the explicit definition. When one method declaration hides another, the return types must be the same. Frankly, there is seldom any good reason to hide a method, and it is generally confusing to readers of a program when you do so. Instance methods with the same parameter types do not hide each other; one *overrides* the other, as described next.

#### 5.8.4 Overriding

When an inherited instance method (i.e., not static) and an explicitly declared instance method have the same name and the same number and types of parameters, we say that the explicitly declared method *overrides* the inherited method rather than hiding it. The return type of the overriding method must be a subtype of the return type of the overridden method (in practice, one usually declares them to return the same type).

When one method hides or overloads another, the two are still distinct methods, unrelated except by name. When one method overrides another, the methods become related. For lack of established terminology, we'll say that they are both members of the same *dynamic method set*. The rules for calling an instance method, as in `x.f(...)` start the same as for static methods: we select which method `f` refers to (if ambiguous) based on the static type of `x`, and on the name (`f`) of the function and the types of the arguments. At this point, things start to differ. First, the call is legal only if `x` is not null (there is an exception otherwise). Next, the method actually called is the member of that method's dynamic method set that is defined (either by inheritance or overriding) in the *dynamic type* of the object referenced by `x`.

The fairly simple statement of this rule hides some rather non-intuitive consequences, but it is the basis of what is called “object-oriented” behavior. Let's look at what it means. Consider the following declarations.



```

class Actor {
    static void print (String s) {
        System.out.println (s);
    }
    static void whereAmI () { print ("in Actor"); }
    String whatAmI () { return "Actor"; } // (1)
    void act () { // (2)
        print ("whatever a(n) " + this.whatAmI () + " does.");
    }
}

class Pinger extends Actor {
    static void whereAmI () { print ("in Pinger"); }
    String whatAmI () { return "Pinger"; } // (3)
    void act () { print ("ping"); } // (4)
}

class Ponger extends Actor {
    static void whereAmI () { print ("in Ponger"); }
    String whatAmI () { return "Ponger"; } // (5)
}

```

In this example, declarations (1), (3), and (5) form one dynamic method set, and declarations (2) and (4) form another. If we call the following function:

```

void doThings () {
    Actor anActor, anotherActor;
    Pinger aPinger;
    Ponger aPonger;
    aPinger = new Pinger ();
    aPonger = new Ponger ();
    anActor = aPinger; anotherActor = aPonger;
    aPinger.whereAmI (); aPonger.whereAmI ();
    anActor.whereAmI ();
    aPinger.act (); aPonger.act ();
    anActor.act (); anotherActor.act ();
}

```

here is what gets printed:

```

in Pinger
in Ponger
in Actor
ping
whatever a(n) Ponger does.
ping
whatever a(n) Ponger does.

```

The first three lines of output are easy to understand. The function `whereAmI` is static. Therefore, we determine which of the three `whereAmI` methods to call by looking at the static types of `aPinger`, `aPonger`, and `anActor`. The result of calling `aPinger.act()` might also seem clear, since the `Pinger` class overrides the `act` method.

The call `aPonger.act()` calls the `act` method inherited from `Actor`, since that is not overridden in `Ponger`. The call to `whatAmI` in `Actor`'s `act` method refers to declaration (1), but since this is an instance method, our rule says that the method that actually gets called is determined by the kind of object that **this** is actually pointing to. Since `aPonger` happens to be pointing at a `Ponger`, instead of calling method (1), we instead call method (5): same dynamic method set, but found by looking in the class of the object currently referenced by `aPonger`.

Again, although the call `anActor.act()` selects declaration (2)—`act` defined in `Actor`—`act` is an instance method, and so the method that is actually called is (4), as determined by the kind of object `anActor` is actually pointing to at the moment. A similar analysis explains `anotherActor.act()`.

**Restrictions on overriding.** The effect of the relationship between methods in a dynamic method set is that when you appear to be calling a method in the set that is defined in one class, you may actually end up calling any of the other methods in the set from classes that override it. For example, if `x` is of type `P`, then `x.f(3)` *looks like* a call to the method `f` defined in class `P`, but, depending on the value of `x`, it could instead call `C.f`, where `C` is a subclass of `P`. But the legality of the parameters you pass to `f` and of the type of the return value (among other things) are determined from the method you appear to be calling, that is, from the declaration of `P.f`. Therefore, there are certain things about the declarations of methods that override each other that have to agree. If method `f` in type `C` overrides `f` in class `P` (that is, `C` extends or implements `P` and `f` has the same number and types of parameters in each), then

- Both `P.f` and `C.f` must be non-static;
- The return type of `C.f` must be a subtype of that of `P.f`.
- `C.f` must be at least as accessible as `P.f` (see §5.11);
- The exception types covered by the **throws** list for `P.f` must include all those covered by the **throws** list in `C.f` (if any).

The guiding principle here is “no surprises.” Since when you call `P.f`, you might get `C.f`, you want the return type and possible exceptions declared for `P.f` to describe what you get back from the call. Likewise, if `C.f` would be illegal due to access restrictions, you don’t want there to be a “back door” for calling it.

## 5.9 Constructor Declarations

A *constructor* is a non-inherited, unnamed function that is called in a restricted way and adorned with special syntax.

### Syntax

*ConstructorDeclaration:*

*ConstructorModifier*\* *SimpleName* *Formals* *Throws*<sub>opt</sub>  
 { *ExplicitConstructorCall*<sub>opt</sub> *BlockStatement*\* }

*ConstructorModifier:*

**public** | **protected** | **private** | *Annotation*

*ExplicitConstructorCall:*

**this** ( *Expression*\* ) ;  
**super** ( *Expression*\* ) ;

The *SimpleName* in a constructor *must* be the name of the class that contains it. This is just a syntactic convention (borrowed from C++); the name is not, properly speaking, the “name of the constructor”—calling `x.C(...)` in a class `C` will simply call an ordinary method named `C`<sup>11</sup>, not the constructor.

**Semantics.** Whenever you create a new object (other than an array) of some type, *T*, using the syntax

```
new T(E1, ..., En)
```

Java’s first operation upon the newly created object is to call the appropriate constructor, as if you had written something like

```
new_object = create a new T;  
new_object.<init>(E1, ..., En);
```

where I use `<init>` to denote the (unmentionable) name of the constructor function. Ordinary overloading rules apply, with the proviso that constructors are not inherited. Constructors don’t allocate new objects; they are intended to give you a chance to initialize them.

The body of the constructor works just like that of an ordinary instance method. However, you are allowed to call other constructors in the first statement (only) of the constructor’s body, using the *ExplicitConstructorCall*. If you omit this call, Java automatically inserts as a default

```
super ();
```

---

**Cross references:** *Annotation* 116, *BlockStatement* 168, *Expression* 119, *Formals* 100, *Throws* 100.

<sup>11</sup>Yes, you can use class names for ordinary function names, since there cannot be any confusion between when you are using a class name and when you are calling a function, and since constructor declarations differ from ordinary function declarations in that they have no explicit return type. No, don’t *ever* do this, regardless of the legalities of the matter, unless you are deliberately trying to obscure your program!

The effect of `this( $E_1, \dots E_n$ )` is to call the constructor in the current class that matches the argument list (matching the same constructor that contains this call results, as usual, in an infinite recursion). The effect of `super( $E_1, \dots E_n$ )` is to call the constructor in the direct superclass that matches the argument list.

Every class contains at least one constructor. If (and only if) you don't provide any explicitly in a class named  $C$ , then Java automatically provides the *default constructor* declaration

```
public C() {
    super ();
}
```

The default constructor matches the default `super()` call that Java inserts when you don't have an *ExplicitConstructorCall*.

There is one exception to all these rules: the (built-in) constructor for class `Object`, which has no superclass, does not call any other constructor.

### 5.9.1 Rationale.

The philosophy behind all these rules and defaults is that

1. That the language should guarantee that a constructor “blesses” every object that gets created.
2. When a class  $C$  extends a class  $P$ ,  $P$ 's constructors must be called to initialize the “ $P$  part” of each  $C$ , and since  $P$  “knows” nothing of the existence of  $C$ , this initialization must happen before any initialization specific to  $C$ .

You can see how this philosophy is reflected in the design of Java's constructors:

- If you supply no constructor, Java creates a default constructor automatically (guaranteeing point 1), and all this does is to call the default constructor of the super class (guaranteeing point 2).
- If you explicitly supply a constructor, it must begin with a call to another constructor—either in the same class (`this(...)`) or in the superclass (`super(...)`). If you don't supply such a call, `super()` is inserted. This guarantees point 2: either your constructor goes into a loop, or it eventually calls a constructor for its superclass.

Let's consider an example:

```
class Unit {
    // public Unit () { super (); }           // (1) implicit
}

class Single extends Unit {
    public Single (int x) { this.x = x; }     // (2)
    public Single () { this (0); }           // (3)
}
```

```

    public int getX () { return x; }
    private int x;
}

class Double extends Single {
    public Double (int x, int y)           // (4)
        { super (x); this.y = y }
    public Double (int x) { this(x, 0); }  // (5)
    public Double () { this (0); }        // (6)
    public int getY () { return y; }
    private int y;
}

```

and the statement

```
Double D = new Double ();
```

First, we create a new object of type `Double`, having the two fields `x` and `y`. Next, we call the matching constructor, (6). That constructor contains only a call on the one-argument constructor, (5), which in turn calls the two-argument constructor, (4), with 0s as the two arguments. Constructor (4) first calls constructor (2), which is found in `Double`'s superclass. That constructor implicitly calls the constructor for `Unit`, which is itself implicit, and which does nothing except call the constructor for class `Object` (which also does nothing in particular, for your information). Next, constructor (2) sets the `x` field of the object and returns. Constructor (4) now sets the `y` field. Finally, we return from (4), then (5), and then (6).

This example illustrates a couple of extraneous points. First, I have followed the convention of naming constructor parameters the same as the fields they are destined to initialize. The use of `this.x` allows me to tell Java when I mean the field and when I mean the parameter. Second, because the field `x` is private (see §5.11), it would be *illegal* to set it in `Double`; I *must* use a constructor for `Single` to initialize it.

### 5.9.2 Instance and field initializers

Besides the code you explicitly write into a constructor and the calls to other constructors implicitly inserted by the compiler, constructors also implicitly handle other initialization included in a class.

#### Syntax.

```

InstanceInitializer: Block
FieldDeclaration:
    FieldModifier* Type InitDeclarator+ ;
InitDeclarator: Declarator VarInit

```

The syntax for field declarations above covers only the case of explicitly initialized fields. This section is only concerned with non-static fields. An *InstanceInitializer* may not contain a **return** statement.

**Semantics.** Each non-static *FieldDeclaration* that contains an explicit initialization (*VarInit*), and each *InstanceInitializer*, contributes code to all constructors that explicitly or implicitly start with the constructor call **super(...)**.

Each *VarInit* contributes an assignment to its field, and the *InstanceInitializers* are inserted as is. For example, the class on the left is equivalent to the one shown on the right:

<pre> class A {     int x = 42;     String name;     A (String name) {         super ();         this.name = name;     }      A () {         super ();         this.name = null;     }      A (int x) {         this (" " + x);     }      {         B.register (this);     } } </pre>	<pre> class A {     int x;     String name;     A (String name) {         super ();         this.x = 42;         {             B.register (this);         }         this.name = name;     }      A () {         super ();         this.x = 42;         {             B.register (this);         }         this.name = null;     }      A (int x) {         this (" " + x);     } } </pre>
--	---

Well, almost equivalent; there is one small difference: the instance initializers do not “see” the formal parameters of the constructors they are inserted into. If you write programs where this makes a difference, you are probably being gratuitously obscure.

### 5.9.3 Some Useful Constructor Idioms

**Uninstantiatable classes** have no instances. This is appropriate for a class containing only static methods and fields. The class in this case is intended merely as a convenient packaging construct for this cluster of definitions. It makes no sense

to create instances of such a class. To document and enforce this property, one useful trick (more politely called an *idiom*) is to define only a *private constructor* in the class, like this:

```
public class UsefulStuff {
    /** Non-instantiatable */
    private UsefulStuff () { }

    static void sort (int A[]) { ... }
    ...
}
```

Any attempt outside the class to write `new UsefulStuff()` attempts to call the parameterless constructor, but since this is private, it is not accessible, and the allocation is illegal.

**Singleton classes** have only one instance, appropriately called a *Singleton*<sup>12</sup>. Such a class is useful when a particular object represents something unique within a program, like “the file system.” One way to accomplish this is again to define a class with only private constructors that initializes a field using this constructor:

```
public class OneOf {
    private OneOf (...) { ... }

    public final static OneOf THE_ONE = new OneOf ();

    ...
}
```

The class `OneOf` is, of course, able to call its own constructor. The field `OneOf.THE_ONE` is the one and only object of type `OneOf` that will ever exist; nothing outside the class `OneOf` is allowed to create another.

**Copy constructors** create new objects that are copies of existing instances. They have a form like this:

```
class MyClass {
    ...
    public MyClass (MyClass x) { ... }
    ...
}
```

where the body of the constructor does whatever is appropriate to to cause **this** to be a “copy of” `x`. Typically, this would mean that after

---

<sup>12</sup>The term is often capitalized like this to emphasize that it is the name of a *design pattern*, as the term is defined in *Design Patterns: Elements of Reusable Object-Oriented Software*, by E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Addison-Wesley, 1995.

```
z = new MyClass(y);
```

we will have `z != y` and `z.equals(y)`, or at least, we would expect the object referenced by `z` to behave just like that referenced by `y`.

Many authorities believe that the copy constructor idiom is preferable to overriding the `clone` method (which is defined as a protected method for all classes in class `java.lang.Object`)<sup>13</sup>.

## 5.10 Initialization of Classes

The design of Java anticipates having your program modify itself as it runs by loading classes “on-the-fly,” possibly as a result of computation. One consequence is that classes are not actually initialized until they are first used—more precisely, a type *T* is not initialized until either

- An instance of *T* is created with **new**, or
- A static method of *T* is called, or
- A static field of *T* is assigned to, or
- The value of a static field of *T* is used, and the field is not a final field initialized by a constant expression (§6.9).

When any of these happen, the class is initialized by initializing its static fields and executing its static initializers.

### Syntax.

*StaticInitializer*: **static** *Block*

*FieldDeclaration*:

*FieldModifier*\* **static** *FieldModifier*\* *Type* *InitDeclarator*<sup>+</sup> ;

*InitDeclarator*: *Declarator* *VarInit*

The syntax for field declarations above covers only the case of explicitly initialized fields. A *StaticInitializer* may not contain a **return** statement.

**Semantics.** In effect, all the assignments to static fields (other than assignments of constant expressions to final fields<sup>14</sup>) and all the *StaticInitializers*, in the order in which they appear, are gathered together into a single anonymous static initialization method. This method is then called when a class is initialized.

<sup>13</sup>See, for example, J. Bloch, *Effective Java*<sup>TM</sup>: *Programming Language Guide*, Addison-Wesley, 2001, pp. 45–52.

**Cross references:** *Block* 168.

<sup>14</sup>The reason for this exception is that final fields initialized by constant expressions never have to be made “real” by an implementation. Since compilers “know” what value they have (their initializers being constant expressions and it being impossible to assign to them again), the compiler can always just substitute this value for occurrences of the field.



For example, the class on the left corresponds conceptually to the one shown on the right (where `<clinit>` stands for the anonymous static initialization function):

<pre>class A {     static int x = B.f(42);     int y = 42;      static {         System.out.print ("OK");     }     ... }</pre>	<pre>class A {     static int x;     int y = 42;     static void &lt;clinit&gt; () {         x = B.f(42);         {             System.out.print ("OK");         }     } }</pre>
---	--

In effect, the system calls `<clinit>()` after loading the class.

## 5.11 Access control

One common theme you'll find throughout software engineering is the distinction between *specifications* and *implementations* of those specifications. Certain methods and constructors of a class are intended to be used by *clients* of that class. Others are for *internal* use, part of the implementation of the class that are not intended for outside eyes. Indeed, entire classes may have no other use than as tools used to implement other classes. Collectively, we refer to classes and members that are intended for general use by clients as the *public* or *external specification* of our program.

Corresponding to these concepts, it is common in modern object-oriented languages to provide language features that allow programmers to restrict the availability of some parts of a program component. These features add nothing whatever to what the language allows us to do; they are a sort of enforced documentation of the programmer's intent.

In Java, it is possible to control what parts of a program may mention a given (Java) interface, class, or class member. A class or interface may be declared **public**, which means that that class or interface may be referred to by name in any part of a program. A non-public class or interface may be referred to only within the package in which it is declared. Members of classes may be declared **private**, **protected**, or **public**; otherwise, they are said to have *package-private* or *default* access<sup>15</sup>. Roughly, for an item defined in class *M*,

- Private access is limited to the declaration of class *M*,
- Package-private access additionally includes all classes in the same package as *M*,
- Protected access additionally includes all subclasses of *M*, and

---

<sup>15</sup>The official term is *default access*. I agree with those, like Joshua Bloch, who prefer the more descriptive *package-private*.

- Public access is unlimited.

That is, each kind of access specification in this list expands the one before it. Figure 5.3 illustrates these specifications in more detail. It uses only methods as examples, but the same applies to fields and constructors.

The one really tricky case involves the rules for **protected** access. As you can see from looking at class **E** in Figure 5.3, the protected method `prot_f` may be accessed as `this.prot_f()` (which is the same as just writing `prot_f()`), but not as `x.prot_f()`. The actual rule is that inside **E**, one can access the protected members of **P1.A** using pointers whose static type is **E** (or some subclass of it). Within the text of **E**, the value of `this` has type **E**, so `this.prot_f()` is legal. However, `x` is not *necessarily* an **E**, so `x.prot_f()` is illegal.

## 5.12 The Java Program

Java programs are collections of type definitions—that is, of definitions of classes (§5.4) and interfaces (§5.5). Programs are split up into *source files* or *compilation units*.

### Syntax.

*CompilationUnit:*

*PackageClause*<sub>opt</sub> *ImportClause*\* *TypeDeclaration*\*

The *PackageClause* indicates what package (§5.1.3) the *TypeDeclarations* belong to. The *ImportClauses* simply provide shortcuts for naming selected types used in the *TypeDeclarations*.

### 5.12.1 Executing a Program

The execution of the program consists of having the *Java virtual machine* (often called “the system” in other languages) call an ordinary function named `main` (sometimes called *the main program*), defined in some class and having the header

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

The official definition of Java does not specify exactly how one specifies in which class to look for `main` or where the arguments come from. Using Sun’s JDK (Java Development Kit), for example, the command line

```
java Foo -o 1 glorp.baz
```

uses `Foo.main` as the main program, and passes it an array of three strings, “-o”, “1”, and “glorp.baz”. This sort of invocation is typical of a *standalone application*, a self-contained program run by user command.

Java has gained a great deal of popularity as a language in which to write *applets*, programs that are shipped over the Internet in response to some action by a browser

package P1;	package P2;
public class A {	public class D {
public void pub_f() { }	void func(P1.A x) {
private void priv_f() { }	x.pub_f();
protected void prot_f() { }	x.priv_f();    // ILLEGAL
void deflt_f() { }	x.prot_f();    // ILLEGAL
}	x.deflt_f();    // ILLEGAL
	}
class B {	}
void func(A x) {	
x.pub_f();	class E extends P1.A {
x.priv_f();    // ILLEGAL	void func(P1.A x) {
x.prot_f();	x.pub_f();
x.deflt_f();	x.priv_f();    // ILLEGAL
}	x.prot_f();    // ILLEGAL
}	x.deflt_f();    // ILLEGAL
	this.pub_f();
class C extends A {	this.priv_f(); // ILLEGAL
void func(A x) {	this.prot_f();
x.pub_f();	this.deflt_f();// ILLEGAL
x.priv_f();    // ILLEGAL	}
x.prot_f();	}
x.deflt_f();	
this.pub_f();	
this.priv_f(); // ILLEGAL	
this.prot_f();	
this.deflt_f();	
}	
}	

Figure 5.3: Illustration of the meanings of access qualifiers **public**, **private**, and **protected**.

and then executed by that browser. In this case, things are bit more complicated. The main program is provided by the browser and the applet itself contains a class with a rather more complicated interface. You are still writing a program in this case, but it is one that runs in a special *framework* that is suitable for graphical user interfaces.

### 5.13 Annotations

The various kinds of modifier presented elsewhere in this chapter describe various properties of the things declared in the declarations they modify. With Java 2, version 1.5, it became possible to extend this set of properties pretty much arbitrarily by defining and applying *annotations*. You may decorate basically any kind of declaration with annotations that supply compile-time information (or *meta-data*) that the compiler and other parts of your program may query. We're not going to go into much detail here, except to give some syntax and describe a few pre-defined annotations.

#### Syntax.

*Annotation:*

`@ TypeName AnnotationArgumentsopt`

*AnnotationArguments:*

`( ElementValue*, )`

`( ElementValuePair*, )`

*ElementValue:*

`Operand`

`Annotation`

`{ ElementValue*, , opt }`

*ElementValuePair:* `Identifier = ElementValue`

So basically, an annotation is just a name or a kind of function call, preceded by an '@' symbol. Each *ElementValue* must be a constant expression (§6.9)—roughly some kind of literal, enumeral, or simple expression or array constructor involving these things and predefined operations. Unlike ordinary calls, annotations may have keyword arguments:

```
@SuppressWarnings(value="unchecked")
ArrayList<String> getList (Object obj) {
    return (ArrayList<String>) obj;
}
```

and they may use a shorthand version of the syntax for specifying array values:

```
@SuppressWarnings({ "fallthrough", "deprecated" })
```

A single argument (in this last example, a single array) without a keyword is short for a single keyword argument with the keyword `value`.

---

**Cross references:** *Identifier* 79, *Operand* 119, *TypeName* 82.

As these two examples illustrate, another shorthand used in annotations is that one-element arrays need not be surrounded in braces, so that

```
@SuppressWarnings(value="unchecked") and
@SuppressWarnings(value={"unchecked"})
```

are equivalent.

In the rest of this section, I'll describe some useful predefined annotations. You can introduce new ones using modified interface declarations (introduced by **@interface** rather than **interface**). However, that capability is more than we need for this book.

### 5.13.1 Deprecated

The marker (parameterless) annotation **Deprecated** indicates that programmers should avoid using a certain definition, typically because that definition is obsolete and likely to be replaced in future versions of the software. The effect is simply to cause the compiler to issue a warning if a program uses the declaration. The Java library uses this annotation extensively:

```
/**
 * Deprecated: As of JDK version 1.1,
 * replaced by <code>setVisible(boolean)</code>.
 */
@Deprecated
public void hide() {
    etc.
}
```

It is good practice, as shown here, to put an explanation into the documentation comment that indicates what one should use instead of **hide**.

### 5.13.2 Override

This is another marker annotation that indicates that a certain method declaration overrides one in a superclass. Like **@Deprecated**, it serves as documentation; its only effect is to cause the compiler to issue a message if the method does *not*, in fact, do so (which usually indicates a misspelling or some error in the parameter list). For example,

```
class MyIterator<T> implements Iterator<T> {
    @Override
    T next () {
        etc.
    }

    ...
}
```

By using this annotation, you can pick up certain common errors, such as overloading a method when you intended to override it. For example,

```
class MyClass extends Object {  
    ...  
    @Override  
    public boolean equals (MyClass obj) {  
        ...  
    }  
}
```

will warn you that the signature of `equals` does not match that of the definition of `equals` in `Object` (which takes an argument of type `Object`, not `MyClass`).

### 5.13.3 SuppressWarnings

This annotation tells the compiler not to warn about certain things, specified by an array of strings that name the warnings. Here are a few common ones:

**"all"** suppresses all warnings.

**"unchecked"** suppresses warnings about certain type checks that the program will not perform and that may possibly cause errors later in the program. For example, using `ArrayList` rather than `ArrayList<String>` for a list of `Strings` will cause this error.

**"deprecation"** suppresses warnings about use of deprecated features.

**"fallthrough"** suppresses warnings about **switch** statements (see §7.2.2) in which it is possible for control to flow from one branch to another:

```
switch (color) {  
    case Red:  
        x = 3;  
        // At this point, we "fall through" to the Blue case.  
    case Blue:  
        x = 2;  
        break;  
}
```

In general, you should attach such annotations to the smallest possible declaration (don't suppress warnings on an entire class when you only need suppress them for a couple of methods). You should *never* suppress a warning unless there is no other reasonable way to shut the compiler up.

## Chapter 6

# The Expression Language

In programming languages, *expressions* are constructs that *yield values* when executed. We use the term to distinguish things like `x+3`, which presumably produces a number, from *statements*—such as

```
if (x > y) y = x;
```

which does something (has an *effect*), but which produces no specific number or other quantity as a result—and from *declarations*, whose purpose is to introduce new names. The lines between expressions, statements, and declarations tend to be blurred in Java, as in many similar languages. For example, the usual use of an assignment such as `x = y+3` is to change the value of `x`, so it functions as a statement. However, the language specifies that it also has a value; you can, in fact, write `z + (x = y+3)`, using the assignment as a value. Likewise, a declaration such as

```
int x = 3;
```

both introduces the name `x` and, like an assignment statement, modifies its value. Expressions can have effects (called *side effects* in that case, on the grounds that the expression’s main purpose ought to be producing a value). Nevertheless, it is useful to separate expressions from statements when describing a language. In fact, you will sometimes hear mentions of “the *expression language*” of a programming language, meaning that subset of the language comprising the expressions.

### 6.1 Syntactic overview

**Syntax.** Here is the top-level syntax for the Java expression language:

*Expression:*

*Operand*

*Assignment*

*Operand: UnaryExpression*

*BinaryExpression*

*Operand ? Expression : Operand*

*UnaryExpression:*  
*UnaryOperator Operand*  
*Cast*  
*PostfixExpression*  
*BinaryExpression:*  
*Operand BinaryOperator Operand*  
*Operand instanceof Type*  
*Assignment: LeftHandSide AssignmentOperator Expression*  
*LeftHandSide: Name | FieldAccess | ArrayAccess*  
*PostfixExpression:*  
*Primary*  
*PostfixExpression IncrDecr*  
*Primary: Allocator*  
*OtherPrimary*  
*OtherPrimary: Name*  
*This*  
*Literal*  
*( Expression )*  
*ArrayAccess*  
*FieldAccess*  
*Call*  
*StatementExpression:*  
*Assignment*  
*Call*  
*ClassAllocator ;*  
*IncrDecr UnaryExpression*  
*PostfixExpression IncrDecr*  
*UnaryOperator: + | - | ~ | ! | IncrDecr*  
*BinaryOperator:*  
*\* | % | / | + | -*  
*<< | >> | >>>*  
*< | > | <= | >= | == | !=*  
*& | ^ | |*  
*&& | ||*  
*IncrDecr: ++ | --*  
*AssignmentOperator:*  
*= | \*= | /= | %= | += | -= |*  
*<<= | >>= | >>>= | &= | ^= | |=*

**Semantics.** An *Expression* is used for its value. A *StatementExpression* is a kind of *Expression* that may have side-effects and that may be used as a statement, with its value ignored (see §7.4 and the definition of *Statement* at the beginning

---

**Cross references:** *Allocator* 143, *ArrayAccess* 148, *Assignment* 120, *Call* 156, *Cast* 123, *ClassAllocator* 144, *FieldAccess* 144, *Literal* 122, *Name* 82, *This* 158.



of that chapter). A parenthesized expression,  $(E)$ , has the same meaning as  $E$ ; the parentheses serve to explicitly delimit the operands of an expression, overriding precedence and associativity (see §6.1.1).

### 6.1.1 Prefix, Postfix, and Infix Operators

In the Lisp family of languages, the beginnings and ends of all expressions and subexpressions (operands) are clearly marked:

`(* (+ a b) c)`

unambiguously means “compute  $a+b$  and multiply that result by  $c$ .” In what we call *algebraic languages*, a category that includes C, Basic, Fortran, Java, Ada, Pascal, PL/1, Algol, and most other languages you can think of, we use *infix notation* (operator in the midst of the expression) with implicit parentheses. This introduces opportunities for ambiguity.

It is insufficient to say that “if  $E_0$  and  $E_1$  are (sub)expressions, then  $E_0 - E_1$  and  $E_0 * E_1$  are also expressions,” because that bare statement is ambiguous about the meanings of many expressions. For example, by these rules,  $x-y-z$  might validly be interpreted as either  $(x-y)-z$  or as  $x-(y-z)$ , which differ in value, but which both have the form of nested subtractions. The question is one of how to *associate* (or *group*) the variables with operators. Common usage (adopted by Java) is that we choose the first alternative, placing implicit parentheses around  $x-y$ ; we say that the binary (two-operand) subtraction operator *associates left* or is *left associative*. Likewise, the expression  $x-y*z$  might validly be interpreted as either  $(x-y)*z$  or  $x-(y*z)$ , also with different values. Common usage (again used by Java) is to take the second interpretation—to say that  $*$  *has higher precedence* or *groups more tightly* than  $-$ .

This leads to a common way to describe unary (one-operand) and infix operator expressions: we list the possible operators in order of precedence and give the associativity of operators that have the same precedence. Table 6.1 gives this list for Java. Some operators appear twice because they stand for two different operators, depending on context. The unary operators  $++$  and  $--$  can be either *prefix* operators (appearing before the operand) or *postfix* operators (appearing after the operand). The operators  $+$  and  $-$  can be either unary prefix operators or binary infix operators.

**Semantics.** Once the proper grouping of operands with operators is established, either by explicit parentheses or by precedence and association rules, the operands of all expressions described in this section are evaluated from left to right (in a few cases—the operators  $\&\&$ ,  $||$ , and  $?:$ —some of the operands may be skipped)<sup>1</sup>. For

<sup>1</sup>You will sometimes see the questions of evaluation and grouping carelessly muddled together in descriptions such as “multiplications are performed before additions.” As the example here shows, this is inaccurate—indeed, it is as inaccurate in C or Fortran as it is in Java. In an expression such as  $x+y*z$  the point is not that “the multiplication happens before the addition,” but rather that the things that are multiplied are  $y$  and  $z$ , as opposed to  $x+y$  and  $z$ . It may sound like nit-picking, I know, but details are important in programming, and you should develop a habit of precision.

See	Category	Operators
		<i>Highest precedence</i>
§6.6.4, §5.1.2, §6.7	Postfix	<code>□[□]</code> <code>□.name</code> <code>□++</code> <code>□--</code>
§6.7, §6.3.2, §6.4, §6.5.2 §6.2.1	Prefix Cast	<code>++□</code> <code>--□</code> <code>+□</code> <code>-□</code> <code>~</code> <code>!</code> <code>(□)□</code>
§6.3.2, §6.4 §6.3.2, §6.4, §9.2 §6.3.3 §6.5.3 §6.5.3 §6.3.3 §6.3.3 §6.3.3 §6.5.2 §6.5.2 §6.5.4	Infix	<code>*</code> <code>/</code> <code>%</code> <code>□+□</code> <code>□-□</code> <code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code> <code>&lt;</code> <code>&gt;</code> <code>&gt;=</code> <code>&lt;=</code> <code>instanceof</code> <code>==</code> <code>!=</code> <code>&amp;</code> <code>^</code> <code> </code> <code>&amp;&amp;</code> <code>  </code> <code>□?□:□</code>
§6.7	Right Associative	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&gt;&gt;=</code> <code>&lt;&lt;=</code> <code>&lt;&lt;&lt;=</code> <code>&amp;=</code> <code>^=</code> <code> =</code>
		<i>Lowest precedence</i>

Table 6.1: Table of operators, ordered by precedence. All but the last line are left associative. Where present, the `□` symbols indicate the placement of operands.

example, the expression `f(x)+f(y)+f(q)*f(r)` is grouped as `(f(x)+f(y))+(f(q)*f(r))`. Therefore, we first evaluate `f(x)`, then `f(y)`, then the sum of `f(x)+f(y)`, then `f(q)`, then `f(r)`, then the product `f(q)*f(r)`, and finally the sum of `f(x)+f(y)` and `f(q)*f(r)`.

### 6.1.2 Literals

Java's primitive values are divided into the numeric types—**byte**, **char**, **short**, **int**, **long**, **float**, and **double**—and the type **boolean**. Correspondingly, there are *literals* (basically, constants whose text directly (literally) denotes their values) of types **char**, **int**, **long**, **float**, **double**, and **boolean**. In addition, there are two kinds of pointer-valued literals, for strings and the null pointer.

#### Syntax.

*Literal:*

*IntegerLiteral*

*RealLiteral*

*CharacterLiteral*

*StringLiteral*

*BooleanLiteral*  
*NullLiteral*

## 6.2 Conversions and Casts

The types of values in Java are related to each other, and there are well-defined conversions between some pairs of types (the term *coercion* is also used). Some conversions are *implicit*: the language defines that they occur in certain contexts even without any particular syntax. Otherwise, the programmer can call for conversions explicitly where needed by use of a syntactic construct called a *cast*.

A value of any type may be converted to that type (the *identity conversion*). A value of any numeric type may be converted to any other numeric type, but not to the type **boolean** or to any reference type. Primitive types cannot be converted to reference types or vice-versa. A reference value whose dynamic type is  $T$  may be converted to any type  $R$  if  $T$  extends or implements  $R$ . Null values may be converted to any reference type.

Conversions in general convert values of one type to “analogous” values of another. When the target (converted-to) type contains at least all the values of the source (converted-from) type, we call the operation a *widening* conversion. For example, conversion of **bytes** to **ints** is a widening, since every byte value is numerically an **int** value as well. Otherwise, we have a *narrowing* conversion. In addition, conversions to **float** and **double** from integer types are classified as widenings, even though they may lose some significant bits. In general, the implicit conversions are limited to widenings.

### 6.2.1 Explicit conversions

A cast is an explicit conversion<sup>2</sup>.

#### Syntax.

*Cast*: ( *Type* ) *UnaryExpression*

**Semantics.** A cast has the static type given by its *Type*. Consider a cast of expression  $E$  to type  $T$ , where  $E$  has static type  $S$  and dynamic type  $D$ . It is legal (that is, causes no error at compilation time) iff

- $T$  and  $S$  are numeric types, or

---

**Cross references:** *IntegerLiteral* 126, *RealLiteral* 136, *BooleanLiteral* 142 *CharacterLiteral* 128, *NullLiteral* 143, *StringLiteral* 151.

<sup>2</sup>The official syntax is rather more complicated than this, for technical reasons having to do with the implementation of Java parsers. For our purposes, however, this simple form works just as well. The problem has to do with dealing with such expressions as  $(x)-y$ , which could be be casts (convert  $-y$  to type  $x$ ) or could instead be additions, depending on whether  $x$  is a type name. I am going to pretend that this question is resolved by looking to see whether  $x$  denotes a type, even though that isn't really how it's done.

**Cross references:** *Type* 76, *UnaryExpression* 120

- $T$  and  $S$  are both **boolean**, or
- $T$  and  $S$  are both reference types and there is (or there could be defined) a subclass of  $S$  that is also a subclass of  $T$ . (Intuitively, it must not be completely impossible to write a program in which a pointer declared to be of type  $T$  points to something of that has a subtype of  $S$ . For example, if  $S$  is **Object**, then  $(T)E$  is legal for any reference type  $T$ . On the other hand, if  $S$  is type **String**, then **(Exception)** $E$  is illegal, since no matter what class declarations you add, no **String** will ever be an **Exception**.)

Conversions of integer types to other integer types produce the same value, modulo the target type's size (see §6.3). Conversions of any number to types **double** or **float** yield the closest value of the target type. Conversions from types **double** or **float** to integer types first truncate to an integer (throw away the fraction) and then convert as for integer-to-integer conversions. The value NaN (Not a Number) converts to 0, and the infinities convert to the largest or smallest values of the target type.

Only the identity conversion is allowed for type **boolean**.

A conversion of a reference type (when it is legal according to the rules above) causes no change to the value; it just adjusts the static type—the compiler's idea of what the value might be pointing to. However, the conversion does throw an exception (**ClassCastException**) if the value of  $E$  is not **null** (casts of **null** never cause an exception) and the dynamic type  $D$  is not a subtype of target type  $T$ . For example, if variable  $x$  is declared to have type **Object**, then (as indicated above),

(**String**)  $x$

is legal (because **Strings** are **Objects**, so on the surface it *might* work). However, it will throw an exception when evaluated if in fact  $x$  contains, for example, the value **System.out** (a **PrintStream** object), because while that is indeed a kind of **Object**, it is not a **String**.

### 6.2.2 Implicit conversions

When a certain context requires a value of a given type, you can use any expression whose value can be *implicitly converted* to that type. The implicit conversions (also called *widening conversions*) are as follows:

- All identity conversions;
- **byte** to any numeric type;
- **short** to any numeric type other than **byte** and **char**;
- **char** to any numeric type other than **byte** and **short**;
- **int** to **long**;
- Any integer type to **float** or **double**;

- **float** to **double**;
- Any subtype of reference type *T* to type *T*;
- The literal **null** to any reference type.

The right-hand sides of assignments (§6.7), the actual parameters to method calls (§6.8), and the *Expressions* in initializers (§6.6.4, §5.2) are all implicitly converted, if needed, to the appropriate type.

### 6.2.3 Promotions

*Promotions* are numeric widening conversions used in arithmetic operations. The operands of numeric operations are never **bytes**, **shorts**, or **chars**; whenever an operand has one of those types, it is *promoted* to a larger numeric type. Also, one operand of a binary numeric operation may be promoted to *balance* the expression: to give both operands the same type.

Unary promotion applies to unary operators, to the operands of shifts (see §6.3.3), and to array index values (§6.6.4).

- Types **short**, **byte**, and **char** are widened to **int**.
- The identity conversion for all other types.

Binary promotion applies to numeric binary operators other than shifts:

- If one operand has type **double**, the other is widened to type **double**;
- Otherwise, if one operand has type **float**, the other is widened to type **float**;
- Otherwise, if one operand has type **long**, the other is widened to type **long**;
- Otherwise, both operands are widened to type **int**.

## 6.3 Integers and Characters

The Scheme language has a notion of integer data type that is particularly convenient for the programmer: Scheme integers correspond directly to mathematical integers and there are standard functions that correspond to the standard arithmetic operations on mathematical integers. While convenient for the programmer, this causes headaches for those who have to implement the language. Computer hardware has no built-in data type that corresponds directly to the mathematical integers, and the language implementor must build such a type out of more primitive components. That is easy enough, but making the resulting arithmetic operations fast is not easy.

Historically, therefore, most “traditional” programming languages don’t provide full mathematical integers either, but instead give programmers something that corresponds to the hardware’s built-in data types. As a result, what passes for integer arithmetic in these languages is at least quite fast. What I call “traditional” languages include FORTRAN, the Algol dialects, Pascal, C, C++, Java, Basic, and many others. The integer types provided by Java are in many ways typical.

Type	Modulus	Minimum	Maximum
long	$2^{64}$	$-2^{63}$ (-9223372036854775808)	$2^{63} - 1$ (9223372036854775807)
int	$2^{32}$	$-2^{31}$ (-2147483648)	$2^{31} - 1$ (2147483647)
short	$2^{16}$	$-2^{15}$ (-32768)	$2^{15} - 1$ (32767)
byte	$2^8$	$-2^7$ (-128)	$2^7 - 1$ (127)
char	$2^{16}$	0 0	$2^{16} - 1$ (65535)

Table 6.2: Ranges of values of Java integral types. Values of a type represented with  $n$  bits are computed modulo  $2^n$  (the “Modulus” column).

### 6.3.1 Integral values and their literals

#### Syntax

*IntegerLiteral*: *IntegerNumeral IntegerTypeSuffix<sub>opt</sub>*

*IntegerNumeral*: 0

*PositiveDigit* *DecimalDigit*<sup>+</sup>

0 *OctalDigit*<sup>+</sup>

0x *HexDigit*<sup>+</sup>

0X *HexDigit*<sup>+</sup>

*PositiveOctalDigit*: 1 | 2 | 3 | 4 | 5 | 6 | 7

*OctalDigit*: 0 | *PositiveOctalDigit*

*PositiveDigit*: *PositiveOctalDigit* | 8 | 9

*DecimalDigit*: 0 | *PositiveDigit*

*HexDigit*:

*DecimalDigit*

a | b | c | d | e | f

A | B | C | D | E | F

*IntegerTypeSuffix*: 1 | L

An integer literal is an atomic element of the syntax (or *token*); it may not contain whitespace, and must be separated from other tokens that can contain letters and digits by whitespace or punctuation.

**Semantics.** The integral values in Java differ from the ordinary mathematical integers in that they come from a finite set (or *domain*). Specifically, the five integer types have the ranges shown in Table 6.2. As we will see in §6.3.2, arithmetic on Java integers is also different from familiar arithmetic; Java uses *modular arithmetic*: when a computation would “overflow” the range of values for a type, the value yielded is a *remainder* of division by some *modulus*.

Only types **char**, **int**, and **long** specifically have literals. The *IntegerLiterals* above denote values of type **long** if suffixed with a lower- or upper-case letter ‘L’, and otherwise denote values of type **int**. You have your choice of decimal, octal (base 8), and hexadecimal (base 16) numerals. The digits for the extra six values needed for base 16 are denoted by the letters a–f in either upper or lower case. To get values of other types, you can use a cast, as discussed in §6.2. Thus,

```
(short) 15  // is a short value
(byte) 15   // is a byte value.
```

One odd thing about the syntax is that there is no provision for negative numbers. Instead, negative literals are treated as negated expressions elsewhere in the grammar, so that, for example, `-1` is treated as two tokens, a minus sign followed by an integer literal. For your purposes, the effect is just about the same. Unfortunately, because of this particular treatment, the language designers felt obliged to introduce a small kludge. The legal decimal literals range from 0 to 2147483647 and from 0L to 9223372036854775807L, since those are the ranges of possible positive values for the types **int** and **long**. However, as you can see from Table 6.2, this limit would not allow you to write the most negative numbers easily, so as a special case, the two decimal literals 2147483648 and 9223372036854775808L are allowed *as long as* they are the operands of a unary minus. This leads to the really obscure and useless fact that:

```
x = -2147483648;  // is legal, but
x = 0-2147483648; // is not!
```

And as if this weren’t enough, the range of octal and hexadecimal numerals is bigger than that of decimal numerals. The maximum decimal numerals are  $2^{31} - 1$  and  $2^{63} - 1$ , while the maximum octal and hexadecimal numerals have maximum values of  $2^{32} - 1$  and  $2^{64} - 1$ . The values beyond the decimal range represent negative values, as we’ll see in §6.3.2. The reason for this puzzling discrepancy is the assumption that decimal numerals are intended to represent integers used *as* mathematical integers, whereas octal and hexadecimal values tend to be used for other purposes, including as sequences of bits.

## Characters

Character values (type **char**) are non-negative integers, a fact that causes considerable confusion. Specifically, the integer values 0–65535 represent the characters specified by the Unicode character set, version 2.0 (see §9.1.1). While you could, therefore, refer to character values in your program by their integer equivalents, it is generally considered much better (in particular, more readable) style to use *character literals*, which make clear what character you are referring to:

```
if (c == 'a')    is preferable to    if (c == 97)
```

Escape	Unicode	Meaning	Escape	Unicode	Meaning
<code>\b</code>	<code>\u0008</code>	Backspace (BS)	<code>\r</code>	<code>\u000d</code>	Carriage return (CR)
<code>\t</code>	<code>\u0009</code>	Horizontal tab (HT)	<code>\"</code>	<code>\u0022</code>	Double quote
<code>\n</code>	<code>\u000a</code>	linefeed (LF)	<code>\'</code>	<code>\u0027</code>	Single quote
<code>\f</code>	<code>\u000c</code>	Form feed (FF)	<code>\\</code>	<code>\u005c</code>	Backslash

Table 6.3: Escape sequences for use in character and string literals, with equivalent Unicode escape sequences where they are legal. The carriage return, newline, backslash, and single quote characters are illegal in character literals, as are their Unicode escapes. Likewise, return, newline, backslash, and double quote characters are illegal in string literals.

### Syntax.

*CharacterLiteral*: `' SingleCharacter '`  
`' EscapeSequence '`  
*SingleCharacter*: Any Unicode character except CR, LF, `'` or `\`  
*EscapeSequence*:  
`\b | \t | \n | \f | \r | \" | \' | \\`  
*OctalEscape*  
*OctalEscape*: *ZeroToThree* *OctalDigit*<sub>opt</sub> *OctalDigit*<sub>opt</sub>  
*ZeroToThree*: `0 | 1 | 2 | 3`

Here, ‘LF’ and ‘CR’ refer to the characters “linefeed” (also referred to as *newline*) and “carriage return.” See §9.1.1 on how to write arbitrary Unicode characters using the `\u` notation.

The escape sequences are intended to give somewhat mnemonic but still concise representations for control characters (things that don’t show up as printed characters) and for a few other values beyond the limits of the ASCII character set (that is, the subset of Unicode in which almost all Java code is written). They also allow you to insert characters that are not allowed by the *SingleCharacter* syntax: carriage returns, linefeeds, single quotes, and backslashes.

An *OctalEscape* represents a numeric value represented as an octal (base-8) numeral. It differs from an octal *IntegerNumeral* only in that it yields a value of type **char** rather than **int**. Octal escapes are considered moderately passé in Java; `\u` or the other escape sequences are officially preferred. The other escape sequences denote the characters indicated in Table 6.3.

### Examples.

```
'a'           // Lower-case a
'A'           // Upper-case A
' '           // Blank
'\t'          // (Horizontal) tab character
'\011'        // Another version of horizontal tab
'\u03b4'       // Greek lower-case delta
'δ'           // Another way to write delta (if your
```



```

// compiler supports it)
'c' - 'a'    // The value 2 ('c' is 99, 'a' is 97).
'\377'      // Character value 255 (largest octal escape)
'\u00FF'    // Another way to write '\377'
'y'         // Another way to write '\377' (if your
// compiler supports it)

```

### 6.3.2 Modular integer arithmetic

The integral arithmetic operators in Java are addition ( $x+y$ ), subtraction ( $x-y$ ), unary negation ( $-x$ ), unary plus ( $+x$ ), multiplication ( $x*y$ ), integer division ( $x/y$ ), and integer remainder ( $x\%y$ ).

#### Syntax.

*UnaryExpression: UnaryAdditiveOperator Operand*  
*UnaryAdditiveOperator: + | -*  
*BinaryExpression: Operand BinaryOperator Operand*  
*BinaryOperator: + | - | \* | / | %*

**Semantics.** The integer division and remainder operations yield integer results. Division rounds toward 0 (i.e., throwing away any fractional part), so that  $3/2$  is equal to 1 and both  $(-3)/2$  and  $3/(-2)$  are equal to -1. Division and remainder are related by the formula

$$(x / y) * y + (x \% y) \equiv x$$

so that

$$x \% y \equiv x - ((x / y) * y)$$

Working out a few examples:

$$\begin{aligned}
 5 \% 3 &\equiv 5 - ((5 / 3) * 3) \equiv 5 - (1*3) \equiv 2 \\
 -5 \% 3 &\equiv (-5) - (((-5) / 3) * 3) \equiv (-5) - ((-1)*3) \equiv -2 \\
 5 \% (-3) &\equiv 5 - ((5 / (-3)) * (-3)) \equiv 5 - ((-1)*(-3)) \equiv 2 \\
 (-5) \% (-3) &\equiv (-5) - (((-5) / (-3)) * (-3)) \\
 &\equiv (-5) - (1*(-3)) \\
 &\equiv -2
 \end{aligned}$$

Division or remaindering by 0 throws an `ArithmeticException` (see 8).

On integers, all of these operations first convert (promote) their operands to either the type **int** or (if at least one argument is **long**) the type **long**. After this promotion, the result of the operation is the same as that of the converted operands. Thus, even if  $x$  and  $y$  are both of type **byte**,  $x+y$  and  $-x$  are **ints**. To be more specific:

To compute  $x \oplus y$ , where  $\oplus$  is any of the Java operations  $+$ ,  $-$ ,  $*$ ,  $/$ , or  $\%$ , and  $x$  and  $y$  are integer quantities (of type `long`, `int`, `short`, `char`, or `byte`),

- If either operand has type `long`, compute the mathematical result converted to type `long`.
- Otherwise, compute the mathematical result converted to type `int`.

By “mathematical result,” I mean the result as in normal arithmetic, where  $/$  is understood to throw away any remainder. Depending on the operands, of course, the result of any of these operations (aside from unary plus, which essentially does nothing but convert its operand to type `int` or `long`), may lie outside the domain of type `int` or `long`. This happens, for example, with `100000*100000`.

The computer hardware will produce a variety of results for such operations, and as a result, traditional languages prior to Java tended to finesse the question of what result to yield, saying that operations producing an out-of-range result were “erroneous,” or had “undefined” or “implementation-dependent” results. In fact, they also tended to finesse the question of the range of various integer types; in standard C, for example, the type `int` has *at least* the range of Java’s type `short`, but may have more. To do otherwise than this could make programs slower on some machines than they would be if the language allowed compilers more choice in what results to produce. The designers of Java, however, decided to ignore any possible speed penalty in order to avoid the substantial hassles caused by differences in the behavior of integers from one machine to another.

Java integer arithmetic is *modular*: the rule is that the actual result of any integer arithmetic operation (including conversion between integer types) is equivalent to the mathematical result “modulo the modulus of the result type,” where the modulus is that given in Table 6.2. In mathematics, we say that two numbers are identical “modulo  $N$ ” if they differ by a multiple of  $N$ :

$$a \equiv b \bmod N \text{ iff there is an integer, } k, \text{ such that } a - b = kN.$$

The numeric types in Java are all computed modulo some power of 2. Thus, the type `byte` is computed modulo 256 ( $2^8$ ). Any attempt to convert an integral value,  $x$ , to type `byte` gives a value that is equal to  $x$  modulo 256. There is an infinity of such values; the one chosen is the one that lies between  $-2^7$  ( $-128$ ) and  $2^7 - 1$  ( $127$ ), inclusive. For example, converting the values 127, 0, and  $-128$  to type `byte` simply gives 127, 0, and  $-128$ , while converting 128 to type `byte` gives  $-128$  (because  $128 - (-128) = 2^8 = 256$ ) and converting 513 to type `byte` gives 1 (because  $1 - 513 = -2 \cdot 2^8$ ). The result of `100000*100000`, being of type `int`, is computed modulo  $2^{32}$  in the range  $-2^{31}$  to  $2^{31} - 1$ , giving 1410065408, because

$$100000^2 - 1410065408 = 8589934592 = 2 \cdot 2^{32}.$$

For addition, subtraction, and multiplication, it doesn’t matter at what point you perform a conversion to the type of result you are after. This is an extremely important property of modular arithmetic. For example, consider the computation

$527 * 1000 + 600$ , where the final result is supposed to be a **byte**. Doing the conversion at the last moment gives

$$527 \cdot 1000 + 600 = 527600 \equiv -16 \pmod{256};$$

or we can first convert all the numerals to **bytes**:

$$15 \cdot -24 + 88 = -272 \equiv -16 \pmod{256};$$

or we can convert the result of the multiplication first:

$$527000 + 600 \equiv 152 + 600 = 752 \equiv -16 \pmod{256}.$$

We always get the same result in the end.

Unfortunately, this happy property breaks down for division. For example, the result of converting  $256/7$  to a **byte** (36) is not the same as that of converting  $0/7$  to a **byte** (0), even though both 256 and 0 are equivalent as **bytes** (i.e., modulo 256). Therefore, the precise points at which conversions happen during the computation of an expression involving integer quantities are important. For example, consider

```
short x = 32767;
byte y = (byte) (x * x * x / 15);
```

A conversion of integers, like other operations on integers, produces a result of type  $T$  that is equivalent to the mathematical value of  $V$  modulo the modulus of  $T$ . So, according to the rules,  $y$  above is computed as

```
short x = 32767;
byte y = (byte) ((int) ((int) (x*x) * x) / 15);
```

The computation proceeds:

```
x*x --> 1073676289
(int) 1073676289 --> 1073676289
1073676289 * x --> 35181150961663
(int) 35181150961663 --> 1073840127
1073840127 / 15 --> 71589341
(byte) 71589341 --> -35
```

If instead I had written

```
byte y = (byte) ((long) x * x * x / 15);
```

it would have been evaluated as

```
byte y = (byte) ((long) ((long) ((long) x * x) * x) / 15);
```

which would proceed:

```
(long) x --> 32767
32767L * x --> 1073676289
(long) 1073676289L --> 1073676289
1073676289L * x --> 35181150961663
(long) 35181150961663L --> 35181150961663
35181150961663L / 15 --> 2345410064110
(byte) 2345410064110L --> -18
```



**Semantics.** One can look at a number as a bunch of bits, as shown in the preceding section. Java (like C and C++) provides operators for treating numbers as bits. The *bitwise operators*—`&`, `|`, `^`, and `~`—all operate by lining up their operands (after binary promotion) and then performing some operation on each bit or pair of corresponding bits, according to the following tables:

Operation	Operand Bits (L,R)				Operation	Operand Bit	
	(0,0)	(0,1)	(1,0)	(1,1)		0	1
<code>&amp;</code> (and)	0	0	0	1	<code>~</code> (not)	1	0
<code> </code> (or)	0	1	1	1			
<code>^</code> (xor)	0	1	1	0			

The “xor” (exclusive or) operation also serves the purpose of a “not equal” operation: it is 1 if and only if its operands are not equal.

In addition, the operation `x<<N` produces the result of multiplying `x` by  $2^N$  (or shifting  $N$  0’s in on the right). `x>>N` produces the result of shifting  $N$  0’s in on the left, throwing away bits on the right. Finally, `x>>N` shifts  $N$  copies of the sign bit in on the left, throwing away bits on the right. This has the effect of dividing by  $2^N$  and rounding down (toward  $-\infty$ ). The left operand of the shift operand is converted to **int** or **long** by unary promotion (see §6.2.3), and the result is the same as this promoted type. The right operand is taken modulo 32 for **int** results and modulo 64 for **long** results, so that `5<<32`, for example, simply yields 5.

For example,

```

int x = 42;    // == 0...0101010 base 2
int y = 7;     // == 0...0000111

x & y == 2    // 0...0000010 | x << 2 == 168    // 0...10101000
x | y == 47   // 0...0101111 | x >> 2 == 10     // 0...00001010
x ^ y == 45   // 0...0101101 | ~y << 2 == -32   // 1...11100000
~y      == -8  // 11...111000 | ~y >> 2 == -2   // 1...11111110
                        | ~y >>> 2 == 230 - 2
                        |                               // 00111...1110
                        | (-y) >> 1 == -4

```

As you can see, even though these operators manipulate bits, whereas **ints** are supposed to be numbers, no conversions are necessary to “turn **ints** into bits.” This isn’t surprising, given that the internal representation of an **int** actually *is* a collection of bits, and always has been; these are operations that have been carried over from the world of machine-language programming into higher-level languages. They have numerous uses; some examples follow.

### Masking

One common use for the bitwise-and operator is to *mask off* (set to 0) selected bits in the representation of a number. For example, to zero out all but the least significant 4 bits of `x`, one can write

```
x = x & 0xf; // or x &= 0xf;
```

To turn off the sign bit (if `x` is an `int`):

```
x &= 0x7fffffff;
```

To turn off all *but* the sign bit:

```
x &= 0x80000000;
```

In general, if the expression `x&~N` masks off exactly the bits that `x&N` does not.

If  $n \geq 0$  is less than the *word length* of an integer type (32 for `int`, 64 for `long`), then the operation of masking off all but the least-significant  $n$  bits of a number of that type is (as we've seen), the same as computing an equivalent number modulo  $2^n$  that is in the range 0 to  $2^n - 1$ . One way to form the mask for this purpose is with an expression like this:

```
/** Mask for the N least-significant bits, 0<=N<32. */
int MASK = (1<<n) - 1;
```

[Why does this work?] With the same value of `MASK`, the statement

```
xt = x & ~MASK;
```

has the interesting effect of *truncating* `x` to the next smaller multiple of  $2^n$  [Why?], while

```
xr = (x + ((1 << n) >>> 1)) & ~MASK;
// or
xr = (x + ((~MASK>1) & MASK)) & ~MASK;
// or, if n > 0, just:
xr = (x + (1 << (n-1))) & ~MASK;
```

*rounds* `x` to the nearest multiple of  $2^n$  [Why?]

## Packing

Sometimes one wants to save space by packing several small numbers into a single `int`. For example, I might know that `w`, `x`, and `y` are each between 0 and  $2^9 - 1$ . I can *pack* them into a single `int` with

```
z = (w<<18) + (x<<9) + y;
```

or

```
z = (w<<18) | (x<<9) | y;
```

From this `z`, I can extract `w`, `x`, and `y` with

```
w = z >>> 18; x = (z >>> 9) & 0x1ff; y = z & 0x1ff;
```

(In this case, the `>>` operator would work just as well.) The hexadecimal value `0x1ff` (or  $11111111_2$  in binary) is used here as a *mask*; it suppresses (masks out) bits other than the nine that interest us. Alternatively, you can perform the masking operation first, extracting `x` with

```
x = (z & 0x3fe00) >>> 9;
```

In order to change just one of the three values packed into `z`, we essentially have to take it apart and reconstruct it. So, to set the `x` part of `z` to 42, we could use the following assignment:

```
z = (z & ~0x3fe00) | (42 << 9);
```

The mask `~0x3fe00` is the complement of the mask that extracts the value of `x`; therefore `z & ~0x3fe00` extracts everything *but* `x` from `z` and leaves the `x` part 0. The right operand is simply the new value, 42, shifted over into the correct position for the `x` component (I could have written 378 instead, but the explicit shift is clearer and less prone to error, and compilers will generally do the computation for you so that it does not have to be re-computed when the program runs). Likewise, to add 1 to the value of `x`, if we know the result won't overflow 9 bits, we could perform the following assignment:

```
z = (z & ~0x3fe00) | (((z & 0x3fe00) >>> 9) + 1) << 9);
```

Actually, in this particular case, I could also just write

```
z += 1 << 9;
```

[Why?]

## 6.4 Floating-Point Numbers

### Syntax.

*UnaryExpression:* *UnaryAdditiveOperator* *Operand*

*UnaryAdditiveOperator:* `+` | `-`

*BinaryExpression:* *Operand* *BinaryOperator* *Operand*

*BinaryOperator:* `+` | `-` | `*` | `/` | `%`

**Semantics.** Just as it provides general integers, Scheme also provides rational numbers—quotients of two integers. Just as the manipulation of arbitrarily large integers has performance problems, so too does the manipulation of what are essentially pairs of arbitrarily large integers. It isn't necessary, furthermore, to have large rational numbers to have large integer numerators and denominators. For example,  $(8/7)^{30}$  is a number approximately equal to 55, but its numerator has 28 digits and its denominator has 27. For most of the results we are after, on the other hand, we need considerably fewer significant digits, and the precision afforded by

large numerators and denominators is largely wasted, and comes at great cost in computational speed.

Therefore, standard computer systems provide a form of limited-precision rational arithmetic known as *floating-point arithmetic*. This may be provided either directly by the hardware (as on Pentiums, for example), or by means of standard software (as on the older 8086 processors, for example).

Java has adopted what is called IEEE Standard Binary Floating-Point Arithmetic. The basic idea behind a floating-point type is to represent only numbers having the form

$$\pm b_0.b_1 \cdots b_{n-1} \times 2^e,$$

where  $n$  is a fixed number,  $e$  is an integer in some fixed range (the *exponent*), and the  $b_i$  are binary digits (0 or 1), so that  $b_0.b_1 \cdots b_{n-1}$  is a fractional binary number (the *significand*) There are two floating-point types:

- **float**:  $n = 24$  and  $-127 < e < 128$ ;
- **double**:  $n = 53$  and  $-1023 < e < 1024$ .

In addition to ordinary numbers, these types also contain several special values:

- $\pm\infty$ , which represent the results of dividing non-zero numbers by 0, or in general numbers that are beyond the range of representable values;
- $-0$ , which is essentially the same as 0 (they are `==`, for example) with some extra information. The difference shows up in the fact that `1/-0.0` is negative infinity. Used properly, it turns out to be a convenient “trick” for giving functions desired values at discontinuities. Take a course in numerical analysis if you are curious.
- **NaN**, or *Not A Number*, standing for the results of invalid operations, such as `0/0`, or subtraction of equal infinities.

To test `x` to see if it is infinite, use `Double.isInfinite(x)`. One checks a value `x` to see if it is not a number with `Double.isNaN(x)` (you can’t use `==` for this test because a NaN has the odd property that it is not equal, greater than, or less than any other value, including itself!) Because of these NaN values, every floating-point expression can be given a reasonable value.

### 6.4.1 Floating-Point Literals

#### Syntax

*RealLiteral*:

```
DecimalDigit+ . DecimalDigit* Exponentopt FloatTypeopt
. DecimalDigit+ Exponentopt FloatTypeopt
DecimalDigit+ Exponent FloatTypeopt
DecimalDigit+ Exponentopt FloatType
```



*Exponent:*

**e** *Sign<sub>opt</sub> DecimalDigit<sup>+</sup>*

**E** *Sign<sub>opt</sub> DecimalDigit<sup>+</sup>*

*Sign:* + | -

*FloatType:* **f** | **F** | **d** | **D**

As for integer literals, no embedded whitespace is allowed. Upper- and lower-case versions of *FloatType* and of the ‘e’ in *Exponents* are equivalent. Literals that end in **f** or **F** are of type **float**, and all others are of type **double**. There are no negative literals, since the unary minus operator does the trick, as for integer literals.

An *Exponent* of the form ‘E±N’ means  $\times 10^{\pm N}$ , and otherwise these literals are interpreted as ordinary numbers with decimal points. For example, 1.004**e**-2 means 0.01004. However, since the floating-point types do not include all values that one can write this way, the number you write is first *rounded* to the nearest representable value. For example, there is no exact 24-bit binary representation for 0.2; its binary representation is 0.00110011.... Therefore, when you write 0.2**f**, you get instead 0.20000000298....

Floating-point literals other than zero must be in the range

1.40239846**e**-45**f** to 3.40282347**e**+38**f** (**float**)

4.94065645841246544**e**-324 to 1.79769313486231570**e**+308 (**double**)

The classes `java.lang.Float` and `java.lang.Double` in the standard library contain definitions of some useful constants.

**Double.MAX\_VALUE**, **Float.MAX\_VALUE** The largest possible values of type **double** and **float**.

**Double.MIN\_VALUE**, **Float.MIN\_VALUE** The smallest possible values of type **double** and **float**.

**Double.POSITIVE\_INFINITY**, **Float.POSITIVE\_INFINITY**  $+\infty$  for types **double** and **float**.

**Double.NEGATIVE\_INFINITY**, **Float.NEGATIVE\_INFINITY**  
 $-\infty$  of type **double** and **float**.

**Double.NaN**, **Float.NaN** A Not-A-Number of type **double** and **float**.

## 6.4.2 Floating-point arithmetic

In what follows, I am going to talk only about the type **double**. This is the default type for floating-point literals, and in the type commonly used for computation. The type **float** is entirely analogous, but since it is not as often used, I will avoid redundancy and not mention it further. The type **float** is useful in places where space is at a premium and the necessary precision is not too high.

The floating-point arithmetic operators in Java are addition ( $x+y$ ), subtraction ( $x-y$ ), unary negation ( $-x$ ), unary plus ( $+x$ ), multiplication ( $x*y$ ), division ( $x/y$ ), and remainder ( $x\%y$ ). The remainder operation obeys the same law as for integers:

$$(x / y) * y + (x \% y) \equiv x$$

except that division is floating-point division rather than integer division; it doesn't discard the fraction.

The result of any arithmetic operation involving ordinary floating-point quantities is rounded to the nearest representable floating-point number (or to  $\pm\infty$  if out of range). In case of ties, where the unrounded result is exactly halfway between two floating-point numbers, one chooses the one that has a last binary digit of 0 (the rule of *round to even*.) The only exception to this rule is that conversions of floating-point to integer types, using a cast such as `(int) x`, always *truncate*—that is, round to the number nearest to 0, throwing the fractional part away<sup>3</sup>.

The justifications for the round-to-even rule are subtle. In computations involving many floating-point operations, it can help avoid biasing the arithmetic error in any particular direction. It also has the very interesting property of preventing drift in certain computations. Suppose, for example, that a certain loop has the effect of computing

$$x = (x + y) - y;$$

on each of many iterations (you wouldn't do this explicitly, of course, but it may happen to one of your variables for certain particular values of the input data). The round-to-even rule guarantees that the value of  $x$  here will change at most once, and then drift no further (a remarkably hard thing to prove, I'm told, but feel free to try).

So far, we've discussed only the usual cases. Operations involving the extra values require a few more rules. Division of a positive or negative quantity by zero yields an infinity, and  $-0$  reverses the sign of the result. Dividing  $\pm 0$  by  $\pm 0$  yields a NaN, as does subtraction of equal infinities, division of two infinite quantities, or any arithmetic operation involving NaN as one of the operands.

In principle, I've now said all that needs to be said. However, there are many subtle consequences of these rules. You'll have to take a course in numerical analysis to learn all of them, but for now, here are a few important points to remember.

### Binary vs. decimal

Computers use binary arithmetic because it leads to simple hardware (i.e., cheaper than using decimal arithmetic). There is, however, a cost to our intuitions to doing this: although any fractional binary number can be represented as a decimal fraction, the reverse is not true. For example, the nearest `double` value to the decimal fraction 0.1 is

0.1000000000000000055511151231257827021181583404541015625

---

<sup>3</sup>The handling of rounding to integer types is *not* the IEEE convention; Java inherited it from C and C++.

so when you write the literal `0.1`, or when you compute `1.0/10.0`, you actually get the number above. You'll see this sometimes when you print things out with a little too much precision. For example, the nearest `double` number to `0.123` is

`0.12299999999999999822364316...`

so that if you print this number with the `%24.17e` format from our library, you'll see that bunch of 9s. Fortunately, less precision will get rounded to something reasonable.

### Round-off

For two reasons, the loop

```
double x; int k
for (x = 0.1, k = 1; x <= N; x += 0.1, k += 1)
{ ... }
```

(where  $10N < 2^{31}$ ) will not necessarily execute  $10N$  times. For example, when  $N$  is 2, 3, 4, or 20, the loop executes  $10N - 1$  times, whereas it executes  $10N$  times for other values in the range 1–20. The first reason is the one mentioned above: `0.1` is only approximately representable. The second is that each addition of this approximation of `0.1` to `x` may round. The rounding is sometimes up and sometimes down, but eventually the combined effects of these two sources of error will cause `x` to drift away from the mathematical value of  $0.1k$  that the loop naively suggests. To get the effect that was probably intended for the loop above, you need something like this:

```
for (int kx = 1; kx <= 10*N; k += 1) {
    double x = kx * 0.1;
    // or double x = (double) kx / 10.0;
```

(The division is more accurate, but slower). With this loop, the values of `x` involve only one or two rounding errors, rather than an ever-increasing number. Still, IEEE arithmetic can be surprisingly robust; for example, computing  $20 \times 0.1$  rounds to exactly 2 (a single multiplication is more accurate than repeated additions).

On the other hand, since integers up to  $2^{53} - 1$  (about  $9 \times 10^{15}$ ) are represented exactly, the loop

```
for (x = 1.0, k = 1; x <= N; x += 1.0, k += 1) { ... }
```

will execute exactly  $N$  times (if  $N < 2^{53}$ ) and `x` and `k` will always have the same mathematical value. In general, operations on integers in this range (except, of course, division) give exact results. If you were doing a computation involving integers having 10–15 decimal digits, and you were trying to squeeze seconds, floating-point might be the way to go, since for operations like multiplication and division, it can be faster than integer arithmetic on `long` values.

In fact, with care, you might even use floating-point for financial computations, computed to the penny (it has been done). I say “with care,” since `0.01` is not

exactly representable in binary. Nevertheless, if you represent quantities in pennies instead of in dollars, you can be sure of the results of additions, subtractions, and (integer) multiplications, at least up to \$9,999,999,999,999.99.

When the exponents of results exceed the largest one representable (*overflow*), the results are approximated by the appropriate infinity. When the exponents get too small to represent at all (*underflow*), the result will be 0. In IEEE (and Java) arithmetic, there is an intermediate stage called *gradual underflow*, which occurs when the exponent ( $e$  in the formula above) is at its minimum, and the first significant bit ( $b_0$ ) is 0.

We often describe the rounding properties of IEEE floating-point by saying that results are correct “to 1/2 unit in the last place (ulp),” because rounding off changes the result by at most that much. Another, looser characterization is to talk about *relative error*. The relative-error bound is pessimistic, but has intuitive advantages. If  $x$  and  $y$  are two `double` quantities, then (in the absence of overflow or any kind of underflow) the computed result,  $x*y$ , is related to the mathematical result,  $x \cdot y$ , by

$$x*y = x \cdot y \cdot (1 + \epsilon), \text{ where } |\epsilon| \leq 2^{-53}.$$

and we say that  $\epsilon$  is the relative error (it’s bound is a little larger than  $10^{-16}$ , so you often hear it said that double-precision floating point gives you something over 15 significant digits). Division has essentially the same rule.

Addition and subtraction also obey the same form of relative-error rule, but with an interesting twist: adding two numbers with opposite signs and similar magnitudes (meaning within a factor of 2 of each other) always gives an *exact* answer. For example, in the expression `0.1-0.09`, the subtraction itself does not cause any round-off error (why?), but since the two operands are themselves rounded off, the result is not exactly equal to 0.01. The subtraction of nearly equal quantities tends to leave behind just the “noise” in the operands (but it gets that noise absolutely right!).

## Spacing

Unlike integers, floating-point numbers are not evenly spaced throughout their range. Figure 6.1 illustrates the spacing of simple floating-point numbers near 0 in which the significand has 3 bits rather than Java’s 24 or 53. Because the numbers get farther apart as their magnitude increases, the absolute value of any round-off error also increases.

There are numerous pitfalls associated with this fact. For example, many numerical algorithms require that we repeat some computation until our result is “close enough” to some desired result. For example, we can compute the square root of a real number  $y$  by the recurrence

$$x_{i+1} = x_i + \frac{y - x_i^2}{2x_i}$$

where  $x_i$  is the  $i^{\text{th}}$  approximation to  $\sqrt{y}$ . We could decide to stop when the error

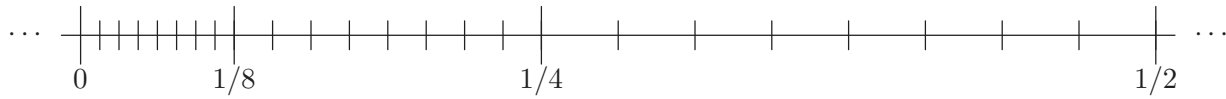


Figure 6.1: Non-negative 3-bit floating-point numbers near 0, showing how the spacing expands as the numbers get bigger. Each tick mark represents a floating-point number. Assume that the minimum exponent is  $-4$ , so that the most closely spaced tick marks are  $2^{-6} = 1/64$  apart.

$|y - x_i^2|$  become small enough<sup>4</sup>. If we decided that “small enough” meant, say, “within 0.001,” then for values of  $y$  less than 1 we would get very few significant digits of precision and for values of  $y$  greater than  $10^{13}$ , we’ll never stop. This is one reason relative error, introduced in the last section, is useful; no matter where you are on the floating-point scale, round off always produces the same relative error.

#### Comment on floating-point equality.

Some textbooks incorrectly tell you never to compare floating-point numbers for equality, but rather to check to see whether they are “close” to each other. This is highly misleading advice (that’s more diplomatic than “wrong,” isn’t it?). It is true that naive uses of `==` can get you into trouble; for example, you should not expect that after setting `x` to `0.0001`, `x*10000==1.0`, since `x` will not be exactly equal to `1/10000`. That simply follows from the behavior of round off, as we’ve discussed.

However, one doesn’t have to be naive. First, we’ve seen that (up to a point) `double` integers work like `ints` or `longs`. Second, IEEE standard arithmetic is designed to behave very well around many singularities in one’s formulas. For example, suppose that  $f(x)$  approaches 0 as  $x$  approaches 1—for concreteness, suppose that  $f(x)$  approximates  $\ln x$ —and that we want to compute  $f(x)/(1 - x)$ , giving it the value 1 when  $x = 1$ . We can write the following computation in Java:

```
if (x == 1.0)
    return 1.0;
else
    return f(x) / (1.0 - x);
```

and it will return a normal number (neither NaN nor infinity) whenever `x` is close to `1.0`. Despite rounding errors, IEEE arithmetic guarantees that `1.0-x` will evaluate to 0 if and only if `x==1.0`.

## 6.5 Booleans and Conditionals

Booleans represent logical values. They have the smallest domain of the primitive types: just the two values `true` and `false`.

<sup>4</sup>In actual practice, by the way, this convergence test isn’t necessary, since the error in  $x_i^2$  as a function of  $i$  is easily predictable for this particular formula.

### 6.5.1 Boolean literals

#### Syntax.

*BooleanLiteral:* **true** | **false**

Boolean literals denote the logical truth values implied by their names, which are the only values of type **boolean**.

### 6.5.2 Boolean operations

#### Syntax.

*UnaryExpression:* ! *Operand*

*BinaryExpression:* *Operand* *BinaryOperator* *Operand*

*BinaryOperator:* & | ^ | | | && | ||

**Semantics.** The operators & (logical and), | (logical or), and ^ (exclusive or) also operate on booleans. The results are the same as for the individual bits of integers subjected to these operators, treating 1 as **true** and 0 as **false**. The ! operator is logical not (analogous to the bitwise complement operator ~). All of these operators yield boolean results.

There are two additional operators that apply only to logical quantities. Conditional and (&&) is the same as & except that it does not evaluate its second operand unless the first is true. For example, the expression

```
x > 0 & 15/x > 3
```

will throw an `ArithmeticException` if `x` is 0, but

```
x > 0 && 15/x > 3
```

will not, since `15/x` is not evaluated if `x` is 0. Likewise, conditional or (||) is the same as | except that it does not evaluate its second operand if the first is true.

The tradition in C, C++, and now Java is that for boolean operations, you use && and || rather than & and |, even when it doesn't matter whether you evaluate both operands. The latter two operators are still useful in bitwise operations (§6.3.3).

### 6.5.3 Comparisons and Equality

#### Syntax.

*BinaryExpression:* *Operand* *BinaryOperator* *Operand*

*BinaryOperator:* < | > | <= | >= | == | !=

---

**Cross references:** *Operand* 119.

**Cross references:** *Operand*“ 119.

**Semantics.** The relational and equality operators compare their operands and produce boolean results. All the operators apply to numeric operands: `<=`, `>=`, `==`, and `!=` are pure ASCII notations for the mathematical operators  $\leq$ ,  $\geq$ ,  $=$ , and  $\neq$  (since `=` has been usurped for assignment, in the great tradition of FORTRAN). For numeric operands, the two operands undergo the standard binary promotions.

The two equality operators, `==` and `!=` apply to all types. However, for the reference types, equality and inequality don't quite mean what you might think. Reference values are pointers, and so these operators mean equality and inequality of *pointers*. That is, if `x` and `y` are references, then `x==y` iff they contain the same pointer value: that is, if they point to the same object. The *contents* of the objects pointed to by `x` and `y` (their fields' values) are irrelevant.

#### 6.5.4 Conditional expressions

##### Syntax.

*Operand: Operand ? Expression : Operand*

**Semantics.** The expression `B?Et:Ef` evaluates `B`, which must be a boolean expression, and then either evaluates and yields the value of `Et` if `B` was true, or `Ef` if `B` was false. Conditional expressions (as opposed to `if...else` statements) are not terribly popular in algebraic languages such as C and Java, as contrasted with languages like Lisp. Indeed, you won't find them in Fortran, Pascal, Ada, and numerous other languages. Frankly, the syntax doesn't help; I recommend keeping them small and using parentheses freely.

The rules for the type of a conditional expression are a bit complicated, since the expressions `Et` and `Ef` may have different (compatible) types. If the types of these expressions are the same, that is the type of the conditional expression. Otherwise, the expression is legal only if one of the operands is assignable (as by an assignment operation) to `T`, the type of the other. In that case, the type `T` is the type of the conditional expression. Thus, `x>y ? "Hello" : null` returns a `String`, since `null` can be assigned to variables of type `String`. Also, `x>y ? 'c' : x+2` has type `int`, if `x` is an `int`.

## 6.6 Reference Types

There are two kinds of literals for the reference (pointer) types, the null literal, which applies to all reference types, and `String` literals. We'll deal with strings in §9.2. The other way to create reference values is to use the operator `new` in an *Allocator*.

##### Syntax.

*NullLiteral: null*

*Allocator: ClassAllocator | ArrayAllocator*

---

**Cross references:** *Expression* 119, *Operand* 119.

As described in §4.1.2, all reference types include a distinguished *null value*, appropriately denoted **null** in Java. It is the default initial value for any container with a reference type.

### 6.6.1 Allocating class instances

The **new** operator creates objects. We'll deal with array objects in §6.6.4. For other objects,

#### Syntax.

*ClassAllocator*: *OuterObject*<sub>opt</sub> **new** *ClassType* ( *Expression*<sup>\*</sup>, )  
*OuterObject*: *Operand* .  
*AnonymousClassAllocator*

We'll deal with anonymous classes in §5.7.3 and deal with the more common case here.

**Semantics.** The effect of a *ClassAllocator* is to create a new object of the named type, and then to call the constructor for that class that matches the arguments (the list of *Expressions*), as for an ordinary method call. Constructors are special functions whose purpose is to initialize an object once it is allocated. Before the constructor is called, the fields have their default values, 0 for all numeric fields, **false** for all boolean fields, and **null** for all fields with reference types. See §5.9 for more details.

The second form is for allocating instances of inner classes. The *Operand* must evaluate to a non-null reference value, *X*, (else `NullPointerException`) and the *SimpleName* must be the name of an inner class defined in the static type of this reference. The resulting instance is associated with *X* as described in §5.7.

### 6.6.2 Field Access

Field accesses extract fields (instance and class variables) from objects.

#### Syntax

*FieldAccess*:  
*Operand* . *Identifier*  
**super** . *Identifier*

There are a couple of syntactic anomalies here. First, since *Operands* can be *Names*, there is a redundancy here with the presence of the *Qualifier* syntax presented in §5.1.2. We're going to take *FieldAccesses* to be restricted to cases where the thing to the left of the dot represents some kind of value, as opposed to a type or package.

---

**Cross references:** *ArrayAllocator* 150, *ClassAllocator* 144.

**Cross references:** *AnonymousClassAllocator* 97, *ClassType* 76, *Expression* 119, *Operand* 119, *SimpleName* 82.

**Cross references:** *Identifier* 79, *Operand* 119.

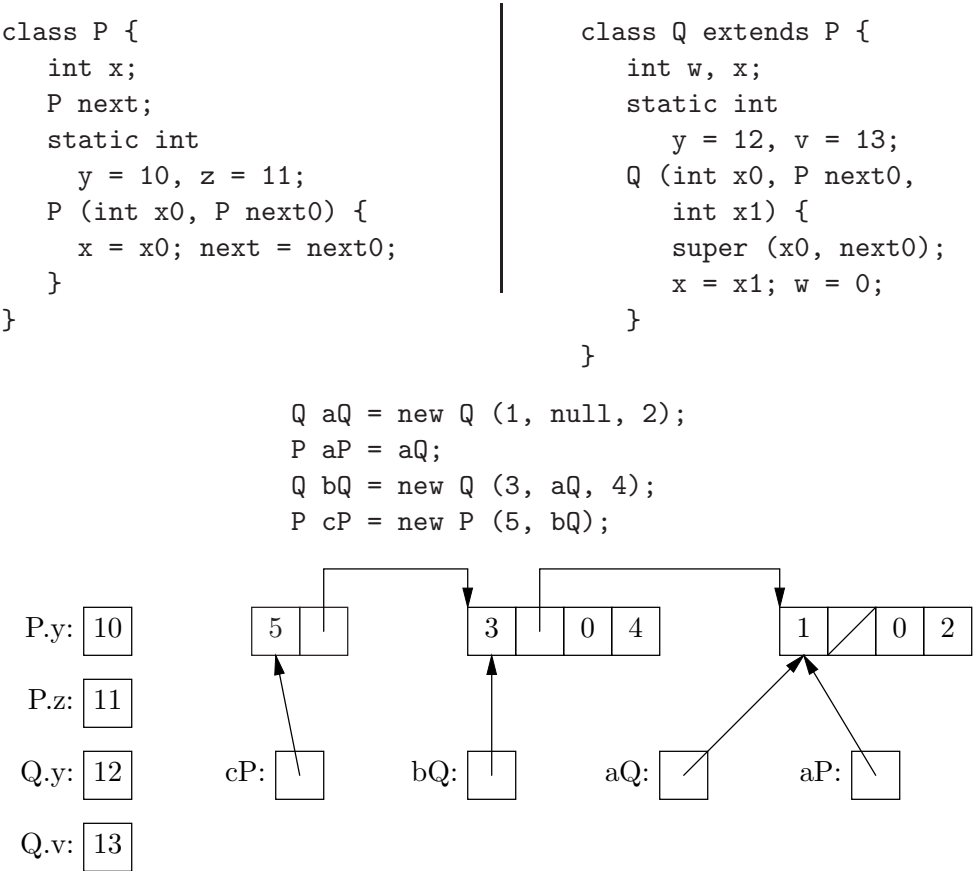


Second, since selection has highest precedence, the *Operand* in a *FieldAccess* can't be just anything. For example, the expression `a+b.length` means `a+(b.length)`.

**Semantics.** Field accesses denote containers. The *Expression* to the left of the dot must evaluate to a reference. If *C* is the static type (reference to class or interface, see §4.2) of this reference, then the *Identifier* to the right of the dot must be the name of a (possibly inherited) field defined in that class or interface. If the field is an instance variable (see §5.4.4) as opposed to a static field, and the reference is null (that is, you ask Java to extract a field from the non-existent “null object”), then the program will implicitly throw the exception `NullPointerException`.

If class *C* extends a class *P*, then the expression `super.x`, when used in an instance method of *C*, means “the value that `this.x` would have in an instance method of *P*.” As a result, it gives access to fields in the superclass of *C* that might be hidden in *C* itself. If a field *x* is defined in *P* (either directly or through inheritance), and *C* hides the declaration of *x*, then `super.x`, refers to this hidden field. The practice of hiding fields of one's superclass is sufficiently questionable that you are not going to see very much about **super** on fields here.

Well, this needs illustration. Figure 6.2 (page 146) shows a couple of class declarations (admittedly contrived) for a class and its superclass, plus an assortment of variables and initializations. The diagram below that shows the state of local variables and objects that results. Finally, the table at the end shows the result of a number of possible field accesses. Some of these results may look counter-intuitive. For example, since `aP` actually points to a `Q` object—the same one as `aQ` points to, in fact—why don't `aP.x` and `aQ.x` fetch the same value? The answer to this and all such questions is actually easier than one might suppose from trying simply to memorize the table: the legality of all field accesses, and the identity of fields they actually fetch, depend entirely on the *static* type of whatever they are selected from (the thing to the left of the dot). It is the fact that `aP` is *declared* to be of type `P` that determines what field `aP.x` references. The rule is simple, but the results can be confusing, which is why it is generally considered bad practice to use the same name for a field in a class as one that is defined in its superclass.



E	Values of E.F				
	.x	.y	.z	.v	.w
aQ	2	12	11	13	0
aP	1	10	11	?	?
bQ	4	12	11	13	0
cP	5	10	11	?	?
P	?	10	11	?	?
Q	?	12	11	13	0
cP.next	3	10	11	?	?
cP.next.next	1	10	11	?	?
bQ.next	1	10	11	?	?
((Q) bQ.next)	2	12	11	13	0
((P) bQ)	3	10	11	?	?
aQ.next	NP	10	11	?	?

Figure 6.2: Example of field extraction. The program text at the top sets up the local variables and objects shown in the diagram below it. Each P object contains an x and next field, each Q object contains both of these, plus (to their right in the diagram) a w field followed by another x field. The table shows the values of various field expressions. A ‘?’ indicated an illegal expression (rejected by the compiler) and a ‘NP’ indicates an expression that will raise a NullPointerException on execution.

### 6.6.3 Testing types

The **instanceof** operator tests the dynamic type of a reference value.

#### Syntax.

*BinaryExpression*: *Operand* **instanceof** *Type*

**Semantics.** A variable that has a reference type may in general point to any of an arbitrary number of different types of object—anything that extends or implements the type of the variable. The **instanceof** operator tests to see whether a certain value actually is pointing to a particular kind of object. Suppose that we have the following declarations

```
class A { ... }
interface I { ... }
class B extends A { ... }
class C extends A implements I { ... }
```

and variables

```
A x = new B (...);
A y = new C (...);
I z = new C (...);
A q = null;
```

Then the following table gives the various of  $v$  **instanceof**  $T$  for all combinations of variable  $v$  and type  $T$ :

$v$	$T$			
	A	I	B	C
x	true	false	true	false
y	true	true	false	true
z	true	true	false	true
q	false	false	false	false

As you can see, the declared types of the variables is independent of the values of **instanceof**. The value **null**, since it does not point to any object, always gives **false** as a result. The type  $T$  is required to be a reference type (possibly an array type, but not a primitive type), and the value of the expression  $v$  must also have a reference type. Furthermore, it is required that it be possible for  $v$  to point to something of type  $T$ ; for example, something like `"A string" instanceof int[]` would be illegal, since a string cannot possibly be an array.

As a general rule, be suspicious of all uses of **instanceof**. In the vast majority of cases, it is not necessary to inquire as to what type of object a variable actually is pointing to. That is, in good object-oriented programming, you will very seldom see code like this:

---

**Cross references:** *Operand* 119, *Type* 76.

```

if (x instanceof Type1) {
    do the Type1 thing to x;
} else if (x instanceof Type2) {
    do the Type2 thing to x;
} ...

```

Instead, `Type1` and `Type2` will both be subclasses of some other class that defines a method (say `doYourThing`), overridden by `Type1` and `Type2`, and you will replace all the code above with just

```
x.doYourThing(...);
```

### 6.6.4 Arrays

An array is a kind of structured container (see §4.1.1). Its inner containers consist of

- a sequence of simple containers that are indexed by the integers  $0$ – $L-1$ , where  $L \geq 0$  is called the length of the array, and
- one constant (**final**) field named `.length`, whose value is always  $L$ , the number of elements in the array.

Array allocators create arrays, and array indexing accesses the indexed elements.

#### Array Indexing

Array accesses denote the individual elements of an array<sup>5</sup>:

#### Syntax.

*ArrayAccess*: *OtherPrimary* [ *Expression* ]

**Semantics.** If the static type of an expression  $A$  is an array type,  $T[]$ , and expression  $E$  is an **int**, **char**, **short**, or **byte**, then  $A[E]$  is the element of the array indexed by value  $E$ , and its static type is  $T$ . If  $A$  is **null**, the program throws `NullPointerException`. If either  $E < 0$  or  $E \geq A.length$  are true, the program throws `ArrayIndexOutOfBoundsException`.

As usual, because the legality rules rely on static types, the following code is illegal, even though  $A$  does indeed reference an array:

```

Object A = new int[5];
A[3] = 2;

```

---

<sup>5</sup>The syntax here is a little picky for a reason. An *OtherPrimary* is a type of simple *Expression* that does not include **new** operators (unless parenthesized). This is necessary because otherwise, **new int[4][3]** is ambiguous: does it mean the last element (#3) of a four-element array just created, or does it mean we are creating an array four arrays of three elements each?

**Cross references:** *Expression* 119, *OtherPrimary* 120.

Arrays are `Objects`, but not all `Objects` are arrays, so the compiler will reject the above, insisting instead that you write

```
Object A = new int[5];
((int[]) A)[3] = 2;
```

—that is, that you cast `A` back to an array type (see §6.2.1 concerning casts).

#### Example: Sum of elements.

```
/** The sum of the elements of A */
int sum(int[] A)
{
    int sum;
    sum = 0;
    for (int i = 0; i < A.length; i += 1)
        sum += A[i];
    return sum;
}
```

#### Example: Circular shift.

```
/** For each k, move the value initially in A[k] to
 *  A[(k+1) % A.length]---that is shift all elements
 *  "to the right" (toward higher indices), moving
 *  the last to the front. */
void shiftRight(int[] A)
{
    if (A.length == 0)
        return;
    int tmp = A[A.length-1];
    for (int i = A.length-2; i >= 0; i -= 1) // Why backwards?
        A[i+1] = A[i];
    A[0] = tmp;
}
```

**Multi-dimensional arrays.** A multi-dimensional Java array is represented as an array of arrays (of arrays...). The syntax accordingly allows you to have multiple indices. For example, if `A` is an array of arrays of **doubles** (type `double[][]`), then the notation `A[i][j]` would first get you element `i` of `A` (which would be an array of type `double[]`), and then from that array would select element `j` (a **double**). If you choose to think of the arrays in `A` as rows of a matrix, then `A[i][j]` would be at column `j` and row `i`.

For example, to get a function that fills an arbitrary two-dimensional array of **doubles** with a value `x`, we can write

```

static void fill (double[][] A, double x) {
    for (int i = 0; i < A.length; i += 1)
        for (int j = 0; j < A[i].length; j += 1)
            A[i][j] = x;
}

```

## Creating arrays

### Syntax.

*ArrayAllocator:*

**new** *Type* *DimExpr*<sup>+</sup> *Dim*\*

**new** *Type* *Dim*<sup>+</sup> *ArrayInitializer*

*Dim:* [ ]

*DimExpr:* [ *Expression* ]

*ArrayInitializer:* { *ElementInitializer*<sup>\*</sup>, *,opt* }

*ElementInitializer:* *Expression* | *ArrayInitializer*

The odd little optional comma at the end of a list of *ElementInitializers* is just a convenience that makes it marginally easier to generate lists of items to paste into one's program.

**Semantics.** Array types are reference types; a declaration such as

```
int[] A;
```

means “at any time, A either contains **null** or a pointer to an array object.” With this particular declaration, A is initially null, as for any variable or field having a reference type. This fact is a rich source of errors for beginning Java programmers:

*WARNING: Declaring an array variable does not put a useful value in it.*

To create an array, we use an array allocator, as in these examples:

```

int[] A = new int[4]; // A points to array of 4 0s.
int[] B;              // Same thing, but in two steps.
B = new int[4];
int[] C = new int[] { 1, 2, 3, 4 }
                      // Array with explicit initial values.
int[] D = { 1, 2, 3, 4 } // Shorthand for the above
sum (new int[] { 1, 2, 3, 4 }) // Pass array value

```

The shorthand form used in the declaration of D is valid only in variable (and field) initializations.

The type of value returned by the array allocator

$$\mathbf{new} \ T[E_1] \cdots [E_m] \underbrace{[ ] \cdots [ ]}_n$$

is  $T[\underbrace{\phantom{[ ] \cdots [ ]}}_{m+n}]$ . None of the values of the  $E_i$  may be negative, or the allocator throws

a **NegativeArraySizeException**. If the resulting array is called  $Z$ , then it has the property that  $Z[i_1][i_2] \cdots [i_m]$  is either a  $T$  (if  $n = 0$ ) or a null pointer to an  $n$ -dimensional array of  $T$ s.

To understand this better, consider the sequence of examples shown in Figure 6.3 (page 152). This shows the effect of various initializations of a two-dimensional array. Example 1 shows the result of no initialization: no array is created, and any attempt to reference `rect[i]` throws an exception. Example 2 creates an array to hold pointers to rows of integers, but the rows themselves are missing: `rect[i]` is OK, but `rect[i][j]` throws an exception. Example 3 finally fills everything in, creating four objects that hold a  $3 \times 4$  array. Example 4 shows that the rows are independent: they need not all have the same length. Example 5 shows that the rows of an array really are general pointers, and that in particular they need not even point to different objects. It changes `rect[0][1]`, but since all rows are the same object, `rect[1][1]` and `rect[1][2]` will also be 42.

### 6.6.5 Strings

The class `java.lang.String` is one of a handful of library classes specifically mentioned in the Java kernel language, where it shows up in the form of *string literals* and the built-in `+` operator on **Strings**.

#### Syntax.

*StringLiteral*: " *StringCharacter*\* "

*StringCharacter*:

Any Unicode character except CR, LF, " or \

*EscapeSequence*

*BinaryExpression*: *Operand* + *Operand*

An object of class **String** represents a sequence of characters (**chars**), and is written with the familiar double-quote syntax:

```
""                // The null string
"Hello, world!"
"δ"              // If supported by your compiler.
```

Null strings are *not* **null** values! They are simply sequences with zero characters in them.

Just as in character literals, escape sequences allow you to insert any character into a string:


```
"Name\tQuantity\tPrice" // Horizontal tabs
"One line.\nAnother line."
// Represents      One line.
//                  Another line.
```

---

**Cross references:** *EscapeSequence* 128, *lhsOperand*.

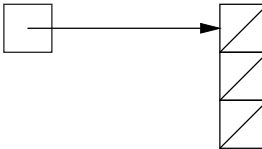
## 1. No initialization

```
int[] [] rect;
```

rect: 

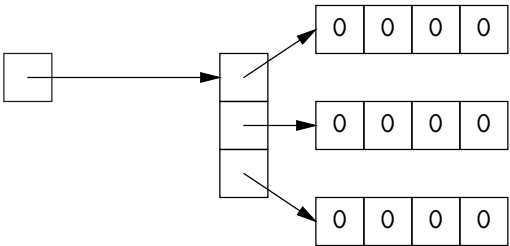
## 2. Initialize one dimension

```
int[] [] rect = new int[3] [];
```

rect: 

## 3. Initialize both dimensions

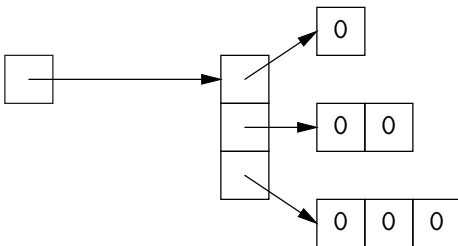
```
int[] [] rect = new int[3][4];
```

rect: 

## 4. Triangular array (two ways)

```
int[] [] triang = new int[3] [];  
for (int i = 0; i < 3; i += 1)  
    triang[i] = new int[i+1];
```

```
/* ---- Another way ---- */  
int[] [] triang =  
    { {0}, {0, 0}, {0,0,0} };
```

triang: 

## 5. Shared row

```
int[] row = { 0, 0, 0 };  
int[] [] rect = { row, row, row }  
rect[0][1] = 42;
```

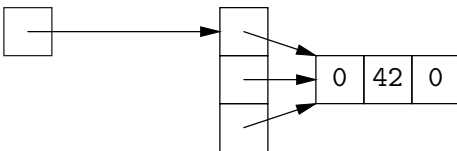
rect: 

Figure 6.3: Illustrations of array initialization.



```
// on UNIX systems.
"2\u03b4 + \u03a8"      // 2δ + Ψ
```

The second example shows one way to insert a line break into a string, but it is not officially portable, since Windows systems, for example, use a different sequence ("`\r\n`"). To really do it right, you should use `System.getProperty("line.separator")` and concatenation, as in

```
"One line." + System.getProperty("line.separator") + "Another line."
```

It is useful to break long strings into pieces for readability. You are not allowed to put bare line breaks into strings, but you can use two literals and the string-concatenation operator (+) to get the desired effect:

```
System.out.println ("Error: too many pending requests. " +
                    "Please try again later.");
```

produces the message

```
Error: too many pending requests. Please try again later.
```

If  $S_0$  and  $S_1$  are `String`-valued expressions,  $X$  is any expression (type `byte`, `int`, `short`, `char`, `long`, `float`, `double`, or `boolean`) whose static type is not `String`, and  $Y$  is any expression<sup>6</sup>,

Expression	Is short for
$S_0 + S_1 \Rightarrow$	$S_0.\text{concat}(S_1)$
$S_0 + X \Rightarrow$	$S_0.\text{concat}(\text{String.valueOf}(X))$
$X + S_1 \Rightarrow$	$\text{String.valueOf}(X).\text{concat}(S_1)$
$S_0 += Y \Rightarrow$	$S_0 = S_0 + Y;$

In the last line, as with other instances of the `+=` operator,  $S_0$  is evaluated only once (see §6.7).

The methods `concat` and `valueOf` are defined in §9.2. The `concat` method concatenates strings, and the `valueOf` methods produce printable external representations of their arguments—for example, `String.valueOf(42)` is `"42"`. The effect of all this is to give you a convenient way to create messages and other strings. For example, to have your program print a `double` value in a variable `x` in the format `x = -1.7`, you can write

```
System.out.println ("x = " + x);
```

The effect of `valueOf` on a reference value is to return either the string `"null"` (for the null value), or to return the result of calling the `toString()` method on that value. Since you can override this method for classes you define, you can arrange that string concatenation produces any desired representation for them. There is an example in Figure 1.3.

<sup>6</sup>I am practicing some revisionism here. In actual fact, the implementations of these operations use the class `StringBuilder`, described in §9.3. However, the effect is the same, which is all that counts.

## 6.7 Assignment, Increment, and Decrement Operators

The core Java language provides only two ways to modify a container: through the assignment operators or the increment and decrement operators.

### Syntax.

*Assignment: LeftHandSide AssignmentOperator Expression*

*LeftHandSide: Name | FieldAccess | ArrayAccess*

*AssignmentOperator:*

`=` | `*=` | `/=` | `%=` | `+=` | `-=`  
`<<=` | `>>=` | `>>>=` | `&=` | `^=` | `|=`

*UnaryExpression: IncrDecr Operand*

*PostfixExpression: PostfixExpression IncrDecr*

*IncrDecr: ++ | --*

The assignment operators are all right associative, so that, for example,

`x = y += z;`      *is equivalent to*   `x = (y += z);`

**Semantics.** The simple assignment operator, `=`, changes the contents of its left operand, which must denote a simple container (a local variable, field, array element, or parameter) to the value yielded by its right operand. It requires that its right operand be implicitly convertible to the type of the left operand (see §6.2). If the left-hand side is a local variable or field, it may not be **final**, unless it is a *blank final*, as described in §6.11.2.

The value of the assignment expression is the final value of the left operand. Together with the right-associativity of assignment, this means that, for example,

`x = y = z;`

assigns the value of `z` to both `x` and `y`.

There is a special shorthand rule to avoid having to write some annoying casts. Consider

```
byte x;
x = 42;
```

Now the value 42 certainly fits in a **byte** (values  $-128$  to  $127$ ), but the static type of 42 is **int**, and type **int** is not implicitly convertible to type **byte**, since not *all* **ints** fit into a **byte**. To avoid having to write

```
x = (byte) 42;
```

when you can see perfectly well that 42 is a possible **byte** value, Java has the rule that in an assignment statement, if the right-hand side is a numeric constant expression (see §6.9) whose (known) value fits in the type of the left operand, then

---

**Cross references:** *ArrayAccess* 148, *Expression* 119, *FieldAccess* 144, *Name* 82, *Operand* 119, *PostfixExpression* 120, *UnaryExpression* 120.

no explicit cast is needed. This exception applies *only* to assignments; if method `f` takes an argument of type `byte`, then `f(42)` is still illegal, and you must write `f((byte) 42)` instead. It also applies only to constant expressions; it would be illegal to write

```
int a = 42;
x = a;      // Illegal
```

The other assignment operators combine a binary operation with an assignment. The expression

$$L \oplus = E$$

—where  $\oplus$  is one of the valid assignment operators listed above,  $L$  is any left-hand side, and  $E$  is any expression—is approximately equivalent to

$$L = (T) (L \oplus E)$$

where  $T$  is the type of  $L$ . Thus,

```
x += 1
```

increments  $x$  by 1, and the value of the entire expression is the resulting value of  $x$ . The left-hand side must not be **final** for these operators.

I said “approximately equivalent” because the container denoted by  $L$  is determined exactly once. This proviso is the sort of obscure pathology that delights language designers. That is, to see why it makes a difference, you have to look at programs that no sane programmer would write. For example,

```
A[i += 1] += 1;
```

If initially,  $i$  is 0, then the effect of this statement in Java is to increment  $i$  to 1, and then increment  $A[1]$  by 1. If we instead wrote,

```
A[i += 1] = A[i += 1] + 1;
```

the effect would be to increment  $i$  *twice*, and then set  $A[1]$  to  $A[2]+1$ <sup>7</sup>.

Historically, C and C++ provided operators specifically tailored for incrementing and decrementing by 1, and Java has retained them:

```
++L    is equivalent to    L += 1
--L    is equivalent to    L -= 1
```

where  $L$  is a left-hand side as before. The corresponding postfix operators,  $L++$  and  $L--$ , affect  $L$  in the same way, but yield as their values the *initial* value of  $L$  (before being incremented or decremented). The operand of  $++$  and  $--$  may not be final.

For obtuse aesthetic reasons, your author never uses these operators ( $+=$  and  $-=$  work just as well and are more general). In most Java code, you will see them used in **for** loops. The post-increment and decrement operators are sometimes used to write obscure but concise statements such as

<sup>7</sup>Java actually defines the outcome here, since it specifies left-to-right evaluation of operands. In C and C++, this statement actually produces unspecified results, either modifying  $A[1]$  or  $A[2]$ , depending on the order in which the two assignments to  $i$  happen.

```
return myStack[sp--];    // Fetch myStack[sp], then decrement
                        // sp just before returning.
```

## 6.8 Method Calls

Method calls encompass what are called function, subprogram, or procedure calls in many other languages, as well as *indirect calls*: calls in which the function actually called is computed on the basis of program data. You should read this section in conjunction with §5.8, which describes the declaration of methods.

### Syntax.

*Call*: *MethodSelector*<sub>opt</sub> *MethodName* *Actuals*  
*MethodSelector*:  
     *ClassType* .  
     *Expression* .  
     **super** .  
*Actuals*: ( *Expression*\* , )  
*MethodName*: *SimpleName*

**Semantics.** The effect of a method call is first to determine the method referred to by the *MethodName*, then to compute the values of the *Actuals* (short for *actual parameters*), to create new local variables, one for each *formal parameter* defined in the declaration of the method, containing copies of the values of the actual parameters (exactly as if by assignment), and finally to execute the body of the selected method.

The *MethodSelector* defines the set of possible method declarations to choose from. If it contains a *ClassType*, then the candidate methods are those defined in (or inherited into) that class. Only static methods may be called this way. If the *MethodSelector* contains an *Expression*, then the candidate methods are those defined in (or inherited into) the class or interface that is the static type of the expression. Either static or non-static (instance) methods may be called this way. An omitted *MethodSelector* is equivalent to ‘**this.**’ (see §6.8.2, below). Finally, if the *MethodSelector* contains **super**, then the candidate methods are those defined in (or inherited into) the superclass of the class that contains the call.

The method actually selected from this candidate set is one with the same *MethodName* as the call. Fields, local variables, types, and statement labels with the same name as the called method are ignored: you can have a field named `foo` and a function named `foo` in the same class without confusion, since the syntax for referring to them is different.

When two or more candidates have the same name (when the method name is *overloaded*), we choose the one declared with the same number of parameters as there are actual parameters and whose formal parameters’ types best match the static types of the actual parameters. That informal description will serve for most

purposes; see §5.8.3 for details. We call the process of selecting a method described there *static method resolution*. For instance methods, there is a further step, described in §6.8.2.

### 6.8.1 Static method calls

For a static method, once the method is selected, the actual parameters are evaluated. Consider a call

`E.f (E1, ..., En)`

and suppose that the declaration of the selected method looks like this:

```
static T f (T1 x1, ..., Tn xn) {
    BODY
}
```

When the method call is used as a statement (that is, ignoring any value it might return), the effect is essentially as if the call were replaced by

```
RETURN:
{
    T1 x1 = v1;
    ...
    Tn xn = vn;
    BODY'
}
```

where the  $v_i$  are the values computed for the actual parameters,  $E_i$ , and `BODY'` is the same as `BODY`, except that any **return** statements (§7.5.3) are replaced by **break RETURN** (I am using **RETURN** to stand for some label that isn't used anywhere else).

The formal parameters of the method, in other words, act like local variables that are initialized to the actual parameters' values. Changing one of these formal parameters in `BODY` has no effect on the value of the actual parameter. For example, if I define

```
static void assign (int x) {
    x = 42;
}
```

and call `assign(y)`, the value of `y` will be unaffected. We say that all parameters in Java are *passed by value*<sup>8</sup>.

---

<sup>8</sup>A number of authors claim that in Java, primitive types are passed by value and reference types are "passed by reference." Avoid this inexcusably sloppy, indeed incorrect, terminology. The actual parameter `y` in this call of `assign` would not be modified even if `y` had a reference type. True, the object that `y` points to can be changed, but in Java it is impossible to pass pointed-to objects, only the pointers to them. The term *pass by reference* refers to languages, like Pascal or (arguably) C++, where one can define functions like `assign` in which assignment to the formal `x` *does* modify the value of actual parameter `y`.

If the call is used as an expression (that is, used for its value), then the static type of the value is the return type declared for the method (the type **T** in the sample declaration above). In this case, the method call is evaluated as just described, except that the value of the method call is that computed in the **return** statement that terminates the call (§7.5.3).

### 6.8.2 Instance Method Calls

Calling an instance method differs from calling static method in two major ways. Otherwise, the treatment of parameters, evaluation of the body, and handling of the return value is the same.

#### The value ‘this’

First, there is an extra, implicit formal parameter, called **this**.

#### Syntax

*This:*  
**this**  
*ClassType* . **this**

**Semantics.** The first, unqualified form is like a **final** parameter; you may not assign to it. The actual parameter value passed to parameter **this** is that of the expression in the *MethodSelector*, (which is the current method’s **this** if the *MethodSelector* is omitted or is **super**). Suppose we have defined

```
class A {
    int y = 0;
    void f (int x) {
        this.y = x;
        y += 1;          // Same as this.y += 1;
        g(3);            // Same as this.g(3)
    }

    void g (int z) {
        y += z;          // Same as this.y += z
    }

    static void h () {
        // this.y = 3; // ILLEGAL
        // y = 3;      // Equivalent to this.y = 3, so ILLEGAL
    }
}
```

and consider

---

**Cross references:** *ClassType* 76.

```
A k = new A();
k.f (3)
```

During the call to `f`, the value of **this** is the same pointer as is contained in `k`, as is also the case during the call to `g` within the body of `f`, so that when the call is finished, `k.y` is 7.

Expressions within the text of an inner class, are allowed to look at the **this** value from enclosing classes. The second, qualified form of **this** allows you to reference these values by specifying which enclosing class you want. See §5.7 for discussion and an example.

Whenever an instance variable or instance method is referred with a “bare” identifier, the appropriate prefix “**this**.” prefix is implicitly prepended.

The keyword **this** is meaningless inside static methods. Therefore, any attempt to refer to the instance variable `y` of **this** in static method `h` would be illegal, as shown.

### Dispatching instance methods

The second difference between instance methods and static methods is in the actual selection of a method to call. Static method selection gives us one candidate method declaration. In the case of instance methods, this method is just one member of a *dynamic method set* consisting of that method declaration plus all method declarations that *override* the declaration (in subclasses of the class that contains the method declaration). All the methods in this set have the same name, parameters, and return type, so any of them could legally be called. There are more details about overriding in §5.8.4.

The actual method that gets called is usually selected from the dynamic method set according to the dynamic type of the *MethodSelector* expression (that is, the dynamic type—the type of the actual pointed-to object—of the value that is passed as the value of **this**). Again, see §5.8.4 for an example. The method call will throw a `NullPointerException` exception if there is no object—that is, if the *MethodSelector* is a null value.

The one exception to this rule is that `super.f(...)`, when it appears in a method in class *C* selects the method to call statically, according to the same rules as `P.f(...)`, where *P* is the superclass of *C*. Otherwise, the value of **super** is essentially **this**, which gets passed as the implicit parameter. This feature is useful when you want to override a method in class *C* so as to augment the effect of the method in *P* rather than to replace it entirely. To extend the example in 6.8.2, if we define

```
class B extends A {
    int q;
    void f (int x) {
        q = x;
        super.f(x);
    }
}
```

```
}
```

the effect is first to save the value of `x` in instance variable `q` and then to increment `y` as does `A.f`.

## 6.9 Constant Expressions

Constant expressions are roughly expressions whose values may be determined independently of any execution of a program. Hence, a compiler can “know” what their values will be, which is necessary for checking certain correctness properties of programs, such as whether the labels of **switch** statements (§7.2) overlap.

### Syntax.

*ConstantExpression*: *Operand*

**Semantics.** A *ConstantExpression* is an operand that contains only the following items:

- Literals.
- Casts to primitive types or to type `String`.
- The unary operators

`+` `-` `!` `~`

- The binary operators

`+` `-` `*` `/` `%` `&` `^` `|` `>>` `<<` `>>>`  
`<` `>` `<=` `>=` `==` `!=` `&&` `||` `?:`

- Names of final variables (local variables and fields) whose initializers are themselves constant expressions.
- Enumerals (see §5.6).

## 6.10 Reflection

Classes define sets of values. We say that 3, for example, is *an int*; we can store it in variables, pass it as a parameter to a function, or return it from a function. On the other hand, you can’t do any of these things to `int` itself, which is a type. For example,

```
void declareSomething (Type typ) {
    typ y;
}
```



and

```
declareSomething (int);
```

are not legal Java. Similarly, these sorts of statements are illegal for methods and packages. We say that types (classes and interfaces), packages, and methods are not *first-class values*. There are languages, such as Lisp or Python, in which these things *are* first-class values, but types, in particular, are seldom first-class in “production” programming languages, for various reasons.

Java does, however, have a facility that comes close, known generally as *reflection*. In programming languages, “reflection” refers to any feature that mirrors entities of a program as values. In Java, values of the class `java.lang.Class` stand for (“reflect”) Java types, but, like other Java values, may be stored, passed as parameters, and so forth. Similarly, there are classes for other kinds of Java entities: `java.lang.reflect.Method` values reflect methods; `java.lang.reflect.Field` values reflect fields; etc. The syntax required to use these reflective values is much clumsier than the corresponding syntax for calling methods or creating instances of types, but one can get many of the same effects. While reflection types are not much used in ordinary programming, they can be important for achieving certain “special effects.” Here, I’ll consider just a few of the features you can obtain using values of type `Class`.

### Syntax.

*Operand:* `Type.class`

For every Java type,  $T$ , there is a value of type `java.lang.Class`, which stands for  $T$  and is otherwise a completely normal Java value. The syntax above provides for literals of type `Class`. In addition, if  $x$  contains a reference value other than `null`, the method call `x.getClass ()`, defined on all `Objects`, will return the `Class` representing its *dynamic* type.

Reflection, unfortunately, is never simple in a language like Java, because of the need to give everything a static type. The class `Class`, in particular, is parameterized (see Chapter 10): The type of `T.class` is actually `Class<T>`. For now, I’ll try to suppress this detail as much as possible, but some of the incantations you see below are necessitated by it.

#### 6.10.1 Type `java.lang.Class`

Given a `Class` value, you can query its methods, fields, containing package, and many other things. For our purposes, however, we’re going to consider just a few operations that `Class` provides—just enough to see how it might be useful.

**The `newInstance` method.** If `cls` has type `Class<C>`, so that `cls` represents the type  $C$ , then then `cls.newInstance ()` returns a new value of type  $C$  as if you had written `new C()`. To actually use it, you must be ready for some exceptional conditions (see Chapter 8). Here’s an example:

---

**Cross references:** *Type* 76.

```

C newObject;
try {
    newObject = cls.newInstance ();
} catch (IllegalAccessException e) {
    newObject = null;
} catch (InstantiationException e) {
    newObject = null;
}

```

The two exceptions here occur if there is no argumentless constructor for the type, or there is a constructor but you are not supposed to be able to use it from this point in the program (it's not public).

Well, this example certainly raises an obvious question: why go to this much trouble, when I could much more easily have written

```
newObject = new C (); ?
```

That works if you know the type `C` when you write this code, but what if you don't? Read on.

**The `forName` method.** The static method `Class.forName` allows you to find a class given its name, even if that class appears nowhere in your program.

Suppose that you are writing a program that, like a typical browser these days, allows users to plug in executable program components as the program executes—components created by some third party, say, to extend the powers of your program, and that perhaps did not exist when you wrote it. A user would tell your program about such a component by typing its name (a `String`), after which your program must somehow add this component to the running program.

Let's further suppose that these components take the form of a Java class. Typically in such situations, you'll have some standard requirements on this package that allow your program to know how to talk to it. One way to impose such requirements is for you to provide a standard interface that the component is to implement:

```

public interface LoadableModule {
    /** Perform any necessary initializations, using ARGUMENTS in a
     *  module-dependent way. */
    void initialize (String[] arguments);
    /** Execute whatever new command this module implements, using
     *  ARGUMENTS as the operands. */
    void execute (String[] arguments);
    /** Shut down this module. */
    void close ();
}

```

Somewhere in your program, you will now need to accept a command that loads modules that conform to this interface. Let's further assume that you want to restrict these modules to like in a particular package.

```

/** Create a module named NAME, which must be the name of a class
 * in package myprogram.modules. */
LoadableModule loadModule (String name) {
    try {
        Class<?> cls = Class.forName ("myprogram.modules." + name);
        return (LoadableModule) cls.newInstance ();
    } catch (ClassNotFoundException e) {
        throw new IllegalArgumentException ("bad module name: " + name);
    } catch (IllegalAccessException e) {
        throw new IllegalArgumentException ("cannot access module " + name);
    } catch (InstantiationException e) {
        throw new IllegalArgumentException ("cannot load module " + name);
    }
}

```

This is an interesting example of object-oriented programming, by the way. We cannot know what modules a user might define, but we can provide a common interface that allows our program to interact with these modules with no knowledge of their internals.

## 6.11 Definite Assignment and Related Obscure Details

Although every container in Java is guaranteed to be initialized to some well-defined value, the language places certain additional requirements, just, I guess, to be on the safe side. Local variables are supposed to be *definitely assigned* before their values are used. In the Java specification, there is an entire chapter devoted to what this means. Frankly, I don't think the precise details matter that much, so I condense the discussion enormously. Intuitively, the basic idea is that it is supposed to be “obvious” that every local variable is assigned to before its value is first used. The test is applied on each method individually (including implicit methods concocted by the compiler for static initializers).

We consider the body of a method and imagine that we change the meaning of all the conditional and looping statements (**if**, **while**, **do**, **for**, and **switch**) so that they are nondeterministic: on any given execution, an **if** might go either way, regardless of its condition; a **while** or **for** might loop or not; and a **switch** might take any branch, regardless of the value of its expression. In addition, imagine that any **catch** clause might get executed at any point in a **try** statement (see Chapter 8). Now, if it is possible that execution of the method under these rules could ever fetch the value of a local variable of the method before it is first assigned to, then the method fails the definite assignment test.

For example, consider

```

int f (int x) {
    int y;
    if (x > 0) {
        y = 3;
    }
}

```

```
    } else {  
        y = 4;  
    }  
    return y;  
}
```

In this case, no matter which branch of the **if** statement is taken, **y** is assigned to before the **return**, so this passes the test.

On the other hand, consider

```
int f (int x) {  
    int y;  
    if (x > 0) {  
        y = 3;  
    }  
    if (x <= 0) {  
        y = 4;  
    }  
    return y;  
}
```

This fails the test, even though you can see that **y** must get assigned to. Since we imagine the **if** statements choosing nondeterministically, ignoring their conditional tests, the definite assignment test requires that **y** get assigned even if *neither* **if** statement executes its assignment, and, of course, **y** does not get assigned under those conditions.

### 6.11.1 Assignments to fields

Fields of records that are initialized with *VarInit* clauses, as in

```
public int x = 14;
```

are also not supposed to be accessed before the assignment takes place (other fields are under no such restriction, and are initialized with the usual default values of 0, **false**, or **null**). This rule can be violated by various pathological programs, generally highly contrived, such as

```
class Weird {  
    int x = y;  
    int y = 42;  
}
```

It turns out, however, that in the general case, it is sometimes impossible to detect this condition at the time a program is compiled. It's still wrong, nevertheless.

### 6.11.2 Blank final variables

It is possible to define a field or local variable to be **final** even though there is no initializer on the declaration. We call these *blank finals*. They are useful when the computation needed to initialize a variable is awkward to write as a simple expression.

Each blank final instance variable must be definitely assigned before the end of every constructor that starts with an explicit or implicit call on the superclass's constructor (see §5.9). Every blank class variable must be definitely assigned before the end of the anonymous static initialization method (see §5.10). No final field or local variable may be assigned more than once. For example, the following class correctly initializes its blank fields:

```
class Blanks {
    static final int x, y;
    final int q;
    int r;

    static {
        x = 42;
        y = x * SomeClass.f ();
    }

    Blanks (int z) {
        q = z;
    }

    Blanks () {
        q = 0;
    }

    Blanks (int z, int w) {
        this (z);
        r = w;
    }
}
```



## Chapter 7

# Statements

In the terminology of programming languages, *statements* are constructs that denote *actions* (as opposed to *expressions*, which denote values, and *declarations*, which define things). A programming language emphasizing statements is known as an *imperative language*, of which Java is an example. As usual, these terms are not as crisply distinguished as I make them appear. In Java, as we'll see, expressions and declarations can denote actions (and act as statements). Nevertheless, the terminology is traditional and often useful.

We can divide statements into two categories: *simple statements* and *control structures*. A control structure is a part of a programming language that is concerned with specifying which statements or expressions in a program get executed and when. Ultimately, a program consists of primitive operations (assignments, arithmetic operations, array indexing, accessing fields of objects, creating objects) contained in simple statements and expressions whose execution is orchestrated by control structures. In this chapter, we'll deal with most control structures, but we'll leave the exception handling to Chapter 8 and concurrent control (*threads*) to Chapter 11.

As a sort of outline of the rest of this chapter, here is the top-level syntax of Java statements:

*Statement: Block*  
*EmptyStatement*  
*LabeledStatement*  
*StatementExpression ;*  
*SelectionStatement*  
*IterationStatement*  
*JumpStatement*  
*TryStatement*  
*SynchronizeStatement*

---

**Cross references:** *Block* 168, *EmptyStatement* 168, *LabeledStatement* 180, *StatementExpression* 120, *SelectionStatement* 169, *IterationStatement* 174, *JumpStatement* 180, *TryStatement* 185, *SynchronizeStatement* 247.

## 7.1 Sequencing, Blocks, and Empty Statements

As you'd expect, writing one statement after another generally means “do the first statement, then do the second.” This control structure (called *sequencing* in statements) is the simplest means of gluing together actions to make more complex actions. In Java, sequences of statements always appear in a syntactic structure called a *block*, which serves to group a sequence of statements into a single statement. This turns out to be extremely useful within other control structures (see §7.2 for the first of many examples).

### Syntax

```
Block: { BlockStatement* }
BlockStatement:
    Statement
    LocalDeclarationStatement
    LocalClassDeclaration
EmptyStatement: ;
```

If you look carefully throughout the chapter at the syntax for other statements (starting from the syntactic outline at the beginning of the chapter), you'll find that they either end with a semicolon, or with another statement. Blocks and class declarations end in “right curly braces” (‘}’). It follows that all statements end in either one or the other, but not both, a useful property to remember.

**Semantics.** Executing a block means executing the sequence of statements and declarations within it in order (except as indicated by certain other control structures; see §8 and §7.5).

As indicated by the Kleene star in the syntax, a block contains zero or more statements. A block containing no statements (written ‘{ }’) does nothing, as does the empty statement (which consists only of a semicolon). You may well wonder at the value of statements that do nothing, but they do occur. Empty blocks, in fact, are somewhat common: all function definitions must have bodies, and it often necessary to give a temporary or default definition, for which “do nothing” is precisely the right specification. The empty statement is in Java more “for completeness” than anything else. One can use it in examples like this:

```
while (mylist.next ())
    ;
```

(meaning “keep calling `mylist.next`, doing nothing else each time, until it returns **false**”). However, the semicolon is so small and easily overlooked that you might be better off using an empty block instead:

```
while (mylist.next ())
{ }
```



## 7.2 Conditional Control

The term *conditional statement* (or to use Java's terminology, *selection statement*) refers generally to anything that selects some (or none) of a collection of possible statements to execute, based on some computed value.

### Syntax.

*SelectionStatement:*

```
if ( Expression ) Statement
if ( Expression ) Statement else Statement
switch ( Expression ) { SwitchBlockStatements* }
```

*SwitchBlockStatements:*

```
SwitchLabel+ BlockStatement+
```

*SwitchLabel:*

```
case ConstantExpression :
default :
```

### 7.2.1 If Statements

An **if** statement '**if** (*C*) *S*<sub>true</sub> **else** *S*<sub>false</sub>' first evaluates its condition (*C*), which must yield a boolean value. It then executes the statement *S*<sub>true</sub> if this value is **true**, and *S*<sub>false</sub> otherwise. A missing **else** part is equivalent to '**else** ;' or '{ }'—"else do nothing."

Since you'll often want to do a sequence of things, not just one, when a certain condition is true, the fact that blocks (§7.1) are statements comes in quite handy:

```
if ( n == 0 ) {
    System.err.println ("Error: list is empty.");
    System.exit (1);
}
```

Here, the single statement *S*<sub>true</sub> is a block containing two statements.

You'll often have a situation in which you want to break a problem into mutually exclusive cases. The **if** statement's syntax fits this quite well:

```
if ( x < 0 )
    System.err.println ("Error: may not be negative.");
else if ( x < 10 )
    handleSmallValue (x);    // Values 0 <= x < 10.
else if ( x < 200 )
    handleMediumValue (x);   // Values 10 <= x < 200.
else
    handleLargeValue (x);    // Values x >= 200.
```

It is conventional to use this indentation style for case analyses, even though everything beginning with the first **else** is technically part of the first **if** statement, everything after the second **else** is part of both the first and second **if** statements, and so forth.

Finally, beware of the subtle ambiguity entailed by the following situation:

```
if (x > 0)
    if (x <= 9)
        handleMediumValue (x);
    else
        handleLargeValue (x);
```

From the bare syntax for `SelectionStatement`, you could interpret this statement either as calling `handleLargeValue` if `x` is greater than 9 or if `x` is less than or equal to 0. The first interpretation assumes that the **else** is associated with the second **if** and the second interpretation assumes it is associated with the first (and the “true” branch is just the immediately following **if** statement without an **else** clause). When there is such a choice, Java resolves this ambiguity by associating such “dangling else” clauses with the nearer **if** statement. If your intentions are different, don’t think that indentation will save you, as in:

```
if (x > 0)
    if (x <= 9)
        handleMediumValue (x);
else
    handleLargeValue (x);           // Oops!
                                   // Executed if x >= 10.
```

This means the same as the preceding example, but is indented so as to fool the reader into thinking otherwise. Instead, use blocks (just as you would use parentheses in an expression):

```
if (x > 0) {
    if (x <= 9)
        handleMediumValue (x);
} else
    handleLargeValue (x);           // Executed if x <= 0
```

### 7.2.2 Switch Statements

One common form of case analysis compares some value for equality to one of a fixed set of values, as in this possible excerpt from a simple calculator program:

```
c = nextInputCharacter ();
if (c == 'q' || c == 'Q')
    quitCommand ();
else if (c == 'p' || c == '=')
    printCommand ();
else if (c == '+')
```

```
        addCommand ();
    else if (c == '-')
        minusCommand ();
    ... // and so on
    else {
        System.err.println ("Invalid command");
        clearInputLine ();
    }
```

Java's **switch** statement provides is a specialized syntax for this kind of situation. With it, the above text might be re-written as

```
switch (nextInputCharacter ()) {
case 'q': case 'Q':
    quitCommand ();
    break;
case 'p': case '=':
    printCommand ();
    break;
case '+':
    addCommand ();
    break;
case '-':
    minusCommand ();
    break;
...
default:
    System.err.println ("Invalid command");
    clearInputLine ();
}
```

The **break** commands in this context mean “leave the **switch** statement.” See §7.5 for a fuller description of **break**.

As you might mostly gather from comparing this **switch** statement with the equivalent **if** statement, the general meaning is this:

1. Evaluate the expression after **switch**. It must have an integral value (in the example above, it is presumably a character—type **char**—which in Java is an integer.) We'll call this value *X* in what follows.
2. Find a switch label (following a **case** keyword) whose value equals *X*. Each switch label must evaluate to an integer (again, in Java character literals evaluate to integers) and must be a constant.
  - 2a. If there is such a switch label, execute the statements that follow up until the closing brace of the **switch** statement, or until otherwise made to stop (typically by a **break** statement).

- 2b. Otherwise, if there is a **default** label (there may be at most one) execute the statements that follow it.
- 2c. Otherwise, leave the **switch** statement.

The switch labels can be any constant expressions (see §6.9), not just simple literals. So, ‘`case X+1:`’ is entirely legal if `X` is a constant, although not if it is an ordinary variable. Because the statements following the cases are `BlockStatements`, they may include declarations. However, there are subtle rules about this, so I advise you to avoid obscurity by wrapping any local declarations you happen to need inside blocks, as in this example:

```
switch (msg.type ()) {
...
case CONTROL_MESSAGE: {    // Start inner block
    String command = msg.extractText (); // Declaration
    ...
    break;
}                          // End inner block
case DATA_MESSAGE:
    ...
}
```

The groups of statements between case labels are known as the *branches* of the **switch** statement. The **break** statement at the end of each branch is a kind of Java idiom marking the end of the branch. If you read the description of how switch statements work closely enough, you can deduce that if you left out a **break** statement—accidentally or otherwise—the effect would be to cause execution of one branch to “fall through” into the next, so that in this example:

```
switch (x) {
case 'Z':
    saveFile ();
    /* FALL THROUGH */
case 'q':
    quit ();
    break;
...
}
```

if `x` contains the character ‘`Z`’, then the program calls both `saveFile` and `quit`. You won’t be tempted to do this very often, and if you do (some tastes dictate that you avoid it), be sure to mark with a comment as shown, so as to avoid puzzling your reader. Aside from this “breakless” situation, the order of your switch labels and the placement of the default branch are irrelevant; choose whatever seems clearest.

## 7.3 About Statement Format

Now that we've seen at least one *structured statement* (a statement containing substatements), it's as good a time as any to take a break to talk about formatting programs for readability. Java, like most modern languages, gives you a great deal of leeway as to how to arrange your program on the page. In general, you can insert extra whitespace (spaces, tabs, line breaks) anywhere. It is traditional to use this freedom to have layout accentuate your programs' logical structure. There is no One True Style for programs that is universally accepted. Nevertheless, it is important for the benefit of those reading your program (yourself included) to adopt consistent conventions for indentation and layout.

In this book, we use a variation of what is sometimes called *K&R style*, after the authors of *The C Programming Language*, Brian Kernighan and Dennis Ritchie. Here are some major points:

1. Within a block, start each statement on a new line.
2. When one line is supposed to be indented relative to the previous line (as indicated by other rules here), indent by some consistent amount. The original K&R convention was four spaces, although two or three are also used<sup>1</sup>.
3. If a statement must continue to a new line, indent the continuation.
4. A block that is part of a larger statement (as for example, an **else** clause that consists of a block) normally has its opening left brace at the end of one line of the larger statement, its block statements indented beginning at the next line, and its closing brace on a separate line, indented at the same level as the larger statement:

```
if (x > y) {
    L = y;
    U = x;
}
```

5. If the true branch of an **if** is a block, place the **else** for that **if** (if there is one) immediately after the closing brace. If the true branch is not a block, put the **else** on a separate line, even with the **if**. Two examples:

<pre>if (x &gt; y) {     L = y;     U = x; } else {     ... }</pre>	<pre>if (x &gt; y)     M = L+1; else     M = U-1;</pre>
---	---

---

<sup>1</sup>Linus Torvalds (inventor of GNU Linux) recommends eight spaces, partially on the grounds that he wants to discourage more than one level of statement nesting. However, such research as has been done on program readability indicates that eight is too much.

6. Unindent lines containing switch labels to the level of the **switch** keyword (see §7.2.2 for examples). The statements between labels are indented as usual. Likewise, unindent the labels of LabeledStatements (see §7.5).

## 7.4 Iteration

There are two ways to repeatedly execute a statement: using *recursive function calls*, and using iterative control structures. Here we consider the latter.

### Syntax

*IterationStatement:*

```
while ( Expression ) Statement
do Statement while ( Expression ) ;
for ( ForInit ForCondition ForUpdateopt ) Statement
for ( finalopt Type SimpleName : Expression ) Statement
```

*ForInit:*

```
StatementExpressionsopt ;
LocalDeclarationStatement
```

*ForCondition:* *Expression*<sub>opt</sub> ;

*ForUpdate:* *StatementExpressions*<sub>opt</sub>

*StatementExpressions:*

```
StatementExpression
StatementExpression ',' StatementExpressions
```

### 7.4.1 While Loops

The four compound statements

```
while (p.next != null)
    p = p.next;

while (i > 0) {
    p = p.next;
    i -- 0;
}

do {
    System.out.println (line);
    line = getNextLine();
} while (line != null);

do
    System.out.println
        (L.next ());
while (L.isMore ());
```

are *loops*. In each case, a statement (which can be a block, as illustrated in two of the examples) is executed alternately with evaluating a condition called the *loop-exit test* (an expression that must yield a boolean value). When the loop-exit test evaluates to false, the loop *terminates*. The difference between the two forms is

---

**Cross references:** *Expression* 119, *StatementExpression* 120, *LocalDeclarationStatement* 85, *Statement* 167.

that the **while** loop starts by first checking its condition, whereas **do...while** first executes its statement before checking the condition for the first time. Thus, the statement in the **do...while** is executed at least once. You will probably find that you don't use this second form very often. Although the choice of where to put the loop-exit test is limited to before and after the statement in these forms, the **break** and **continue** statements provide ways to terminate the loop or the current iteration (i.e., execution of the loop statement) at any time within the statement. Thus we have the common “loop-and-a-half” idiom:

```
while (true) { // Loop "forever"
    c = src.read ();
    if (c == '\n' || c == -1)
        break;          // Stop at end of line or file
    line += c;           // Else add to end of line.
}
```

See §7.5 for more details and examples.

#### 7.4.2 Traditional For Loops

One of the most common kinds of loop executes a statement repeatedly while also assigning some regular sequence of values to a *control variable* used inside the loop. For example, Pascal has a way to say “execute statement *S* repeatedly, first with variable *I* equal to 1, then 2, then 3, up to *N*.” The C language has a generalization of the idea, inherited by C++ and Java and used in numerous scripting languages. It provides what programming-language types call *syntactic sugar* for (that is, a convenient way to write) an equivalent **while** loop.

The general idea is that the loop

```
for (Initial; Test; Update)
    Statement
```

is *almost* equivalent to the block

```
{
    Initial;
    while (Test) {
        Statement
        Update;
    }
}
```

(We'll come to the “almost” a bit later.) If you look at the official syntax at the beginning of §7.4, you'll see that *Test* is optional; an omitted *Test* is equivalent to **true**.

Stated baldly as this equivalence, the meaning of **for** may not be apparent, so let's look at examples. Consider adding all integers between 1 and *N*. Here it is in **for**-loop form next to the equivalent **while**-loop form:

```

int i, S;
S = 0;
for (i = 1; i <= N; i += 1)
    S += i;

int i, S;
S = 0;
{
    i = 1;
    while (i <= N) {
        S += i;
        i += 1;
    }
}

```

You can read the **for** loop as “Starting with *i* set to 1, repeatedly add *i* to *S* and increment *i* by 1 as long as *i* does not exceed *N*.”

I put the equivalent **while** loop in a block in order to explain what happens when you exercise the option given by the syntax to have the initialization (*ForInit*) clause be a declaration. In the example above, we use *i* only inside the **for** loop and have no need for its value afterwards. In such cases, it is good practice to explicitly limit the scope of *i* by writing the loop as follows (again, the translated (or “unsugared”) form is on the right).

```

int S;
S = 0;
for (int i = 1; i <= N; i += 1)
    S += i;

int S;
S = 0;
{
    int i = 1;
    while (i <= N) {
        S += i;
        i += 1;
    }
}

```

The variable *i* is defined only within the **for** loop.

The syntax for *ForInit* indicates that it can contain a comma-separated *list* of *StatementExpressions* (to be executed in the order written). Also, if you look at the syntax of *LocalDeclarationStatement*, you will find that it provides for declaring and initializing a list of variables. We could write our summation loop in either of the following two ways:

```

int S, i;
for (i = 1, S = 0; i <= N; i += 1)
    S += i;

for (int i = 1, S = 0; i <= N; i += 1)
    S += i;

```

Indeed, the *ForUpdate* may also be a list, which allows us to write

```

for (int i = 1, S = 0; i <= N; S += i, i += 1) { }

```

But you may find, as I often do, that this kind of **for** loop looks a bit strange, and the alternative way of writing the empty statement,



```
for (int i = 1, S = 0; i <= N; S += i, i += 1)
    ;
```

is even more unidiomatic (and consider what happens if you forget the semicolon!).

You may have noticed that the last three loops above have another problem. Since each declares **S** to be local to the **for** loop, it is not available outside the loop, so these are not likely to be useful loops as they stand. The Java syntax does not provide a way to both declare **i** and to initialize **S** without declaring it, nor does it provide a way to declare multiple variables with differing types (as when we want **S** to be of type **double**). For these cases, you must use additional declarations or statements outside the loop, as illustrated in several of the examples above.

The examples so far have been **for** loops that work on integer quantities, but there is no restriction on the types of quantity that you can use. For example, to traverse a linked list of integers, (let's call it **aList**), printing all its elements, one might use the following:

```
for (IntList L = aList; L != null; L = L.next)
    System.out.println (L.head);
```

As another example, the standard Java library defines the interface **Iterator** in package **java.util**, which is intended to be a common interface for all kinds of “things that traverse a data structure.” An **Iterator** has operations **next** and **hasNext**, intended to be used like this:

```
import java.util.Iterator;
...
for (Iterator<?> L = aCollection.iterator (); L.hasNext ();) {
    System.out.println (L.next ());
```

That is, **next** is presumably defined to yield “the next item” in some collection of objects, and **hasNext** tells whether there is a next item. The operations of moving on to the next item and of then yielding its value are combined in the **next** function, so there is an empty *ForUpdate* part. The **Iterator** **L** is a sort of “moving finger” pointing into **aCollection**. (We'll get to the meaning of the **<?>** notation in Chapter 10. For now, just think of it as meaning that **L** iterates over collections of *some* kind of **Object**, and it doesn't happen to be important which in this particular loop.)

As this last example illustrates, the syntax allows any of the three parts of the **for** loop to be empty. So, for example, the loop

```
for (; i < N; i += 1)
    ...
```

starts with **i** equal to whatever the previous statements had set it to (such loops, however, might be clearer as **while** loops; it's a matter of taste). An empty *ForCondition* means “true,” so one idiom that you'll sometimes see in place of “**while true**” is

```
for (;;) { ... }
```

### 7.4.3 For Loops for Collections and Arrays

Perhaps the most common use of **for** loops is in sequencing through the elements of some data structure. For example, to add all the elements of an array, *A*, one might write

```
double sum;
sum = 0.0;
for (int i = 0; i < A.length; i += 1)
    sum += A[i];
```

On the other hand, to add all the elements of a standard Java *Collection*, one might write (following the example in §7.4.2):

```
import java.util.Collection;
import java.util.Iterator;
...
static double sum (Collection<Double> aCollection) {
    double sum;
    sum = 0.0
    for (Iterator<Double> i = aCollection.iterator (); i.hasNext (); )
        sum += i.next ();
    return sum;
}
```

So even though arrays and *Collections* are both “a bunch of things,” there are considerably different ways of accessing them.

Along with its other features, Java 2, version 1.5 introduced a new kind of **for** loop specifically for the purpose of iterating over collections of values. The statement

```
for ( $\mathcal{T}$  x :  $\mathcal{E}$ )
    S;
```

converts to one of two different loops, depending on whether the value of expression  $\mathcal{E}$  is an array type or some other kind of object. For an array, we get:

```
{
    // _E0_ and _I_ are names that don't appear in elsewhere in the program.
    C[] _E0_ =  $\mathcal{E}$ ; // Assuming  $\mathcal{E}$  is an array of C.
    for (int _I_ = 0; _I_ < _E0_.length; _I_ += 1) {
         $\mathcal{T}$  x = _E0_[_I_];
        S;
    }
}
```

so that we can replace the first loop with

```
for (double x : A)
    sum += x;
```

For other kinds of object, we get:

```
// Assume that  $\mathcal{E}$  contains objects of type  $C$ .
for (Iterator<C> _I_ =  $\mathcal{E}$ .iterator (); _I_.hasNext (); ) {
    T x = _I_.next ();
    S;
}
```

so that our summation method now looks like this:

```
static double sum (Collection<Double> aCollection) {
    double sum;
    sum = 0.0
    for (Double x : aCollection)
        sum += x;
    return sum;
}
```

—essentially the same loop as for the array.

To qualify as an appropriate type for this loop construct, the type of the expression  $\mathcal{E}$  must either be an array type or must implement the Java library interface type `Iterable`, which is defined like this:

```
package java.util;

public class Iterable<T> {
    Iterator<T> iterator ();
}
```

No surprises here, since the equivalent loop assumes that there is an `iterator` method defined and that it returns an `Iterator`.

## 7.5 Jump Statements and Labels

Numerous programming languages provide an extremely flexible control statement (known generically as the **goto**, after the common keyword) that allowed a program to jump from one statement to any other. However, a note published in 1968<sup>2</sup> started a tradition of avoiding the use of this construct in favor of **if** statements and **while** loops that more clearly indicated the intent behind such jumps. Nevertheless, certain restricted jumps—specifically those that serve to *terminate* the execution some construct—can help to clarify a program. Java, like the language Bliss before it, provides only these restricted jumps.

---

<sup>2</sup>E. W. Dijkstra, “Goto Statement Considered Harmful,” *Communications of the ACM*, 11 (3), 1968, pp. 147–148. The title was actually added by an editor.

**Syntax**

*JumpStatement:*  
     *BreakStatement*  
     *ContinueStatement*  
     *ReturnStatement*  
     *ThrowStatement*  
*LabeledStatement:*  
     *Label* : *Statement*  
*Label:* *Identifier*

A **break** statement is legal only inside a loop, **switch** statement, or a labeled statement with the same label. A **continue** statement is legal only inside a loop, and if it has a label, it must be inside a loop with that label. We'll come to *ThrowStatements* in §8.1.

All jumps cause what is called the *abrupt completion* of some other statements that execute the jumps. That is, execution of these other statements ends immediately, even if they contain later statements that would normally be executed. When a statement completes abruptly, so does the statement that textually contains it (which may be a block, if statement, etc.), and the process repeats until we get to the target of the jump, which depends on the type of jump.

**Statement Labels.** The sole purpose of attaching a label to a statement is to provide a way for **break** or **continue** statements to refer to that statement. As you can see from the syntax, a statement may have any number of labels (since the statement after a *Label* can itself be a *LabeledStatement*), although this is not particularly useful. The label on a statement may not duplicate any other label on or in that same statement. On the other hand, labels *may* duplicate the names of other kinds of things—variables, fields, functions, types, packages, parameters—although as a stylist matter, I strongly advise against this potentially confusing practice. Otherwise, any identifier may be a label; in this book, I use capitalized words.

**7.5.1 The ‘break’ Statement****Syntax.**

*BreakStatement:* **break** *Label<sub>opt</sub>* ;

An unlabeled *BreakStatement* must occur within a **switch**, **while**, **do**, or **for** statement. Otherwise, the *Label* must be attached to a statement that encloses the **break**.

**Semantics.** Executing a **break** statement causes the program to exit from some larger statement that contains the break, and to proceed to the following statement. With a label, **break** exits from the statement with that label. Without a specific

---

**Cross references:** *ReturnStatement* 183 *ThrowStatement* 185 *Identifier* 79

**Cross references:** *Label* 180.

label, it will exit from the *smallest enclosing* loop (**while**, **do**, or **for**) or **switch** statement. That's why it is the standard idiomatic way to mark the end of the branch of a **switch** statement. It also allows us to put loop exits anywhere we want, as illustrated by the loop-and-a-half idiom on page 175.

You'll find breaks with labels useful in exiting from a loop from inside a **switch** statement:

```
GetChars:
    while (true) {
        int c = input.read ();
        switch (c) {
            case -1:
                break GetChars;
            case 'a':
                ...
        }
    }
```

or in exiting from nested loops:

```
/* In the list of arrays L, determine whether any array
 * element, x, satisfies test P. */
Find:
    for (L = myList; L != null; L = L.next) {
        for (int k = 0; k < L.item.length; k += 1)
            if (P (L.item[k])) {
                x = L.item[k];
                break Find; // Stops both loops
            }
    }
```

or when the construct from which you wish to exit is not a loop or **switch**<sup>3</sup>; for example, here we exit from a block:

```
Find: {
    for (k = 0; k < data.length; k += 1) {
        if (P (data[k])) {
            x = data[k];
            break Find;
        }
    }
    for (k = 0; k < reserve.length; k += 1) {
        if (P (reserve[k])) {
            x = reserve[k];
            break Find;
        }
    }
}
```

---

<sup>3</sup>There was a notorious failure in the telephone network around Chicago some time back that was tracked to a C programmer who had incorrectly thought that **break** would exit from an **if** statement. In C, it cannot, which may be why we now find it in Java.

```

    }
}

```

### 7.5.2 The ‘continue’ Statement

#### Syntax.

*ContinueStatement*: **continue** *Label<sub>opt</sub>* ;

A *ContinueStatement* is valid only inside a loop (**while**, **do**, or **for**). If supplied, the *Label* must be attached to a loop that encloses the **continue**.

**Semantics.** Occasionally, one wants to *restart* the statement of a loop rather than to exit from the entire loop. The **continue** statement performs this rather specialized form of jump. The generic **while** and **for** loops on the left have the same effects as the loops on the right:

<pre> while (C) {     ... continue; ... } </pre>	<pre> while (C) {     CONTINUE: {         ...break CONTINUE;...     } // continue here } </pre>
<pre> for (I; C; U) {     ... continue; ... } </pre>	<pre> for (I; C; U) {     CONTINUE: {         ...break CONTINUE;...     } // continue here } </pre>
<pre> do {     ... continue; ... } while (C); </pre>	<pre> do {     CONTINUE: {         ...break CONTINUE;...     } // continue here } while (C); </pre>

That is, **continue** has the effect of terminating the current iteration at any point in the middle and then going back to perform the loop test ( $C_1$ ) and repeat the loop again. (The label *CONTINUE* is intended to represent some label that is unused elsewhere in the program.)

With a labeled **continue**, the effect is the same, except that the loop body that is restarted is the one with the given label. A **continue** statement is only legal in a loop, and the label, if present, must be the label of an enclosing loop.

In truth, **continues** are rather rare in programs. Here, though, is one possible example:

```

while (true) {
    String line = input.readLine ();
    if (line.startsWith("#"))

```

```
        continue; // Comment line. Skip and go to next line.
    Other processing.
}
```

### 7.5.3 The ‘return’ Statement

#### Syntax.

*ReturnStatement*: **return** *Expression<sub>opt</sub>* ;

This statement may occur only inside a method. The *Expression* must *not* be supplied if the method returns **void**, and *must* be supplied otherwise.

**Semantics.** The effect of **return** is to terminate execution of the enclosing method body and return to the caller. If the *Expression* is supplied, it is first evaluated and the method call yields that value (after it is implicitly converted, if necessary, to the return type of the method).





## Chapter 8

# Exception Handling

In the real world, good programs contain a large amount of code whose sole purpose is to check that inputs meet their requirements, to take corrective action when they don't, and to detect and correct error conditions that may arise owing to failures or limitations of one's machine and the hardware it connects to. This is all a vital part of making a program “robust,” “idiot-proof,” and “user-friendly,” and frankly can be extremely dull<sup>1</sup>. It can also clutter up and obscure otherwise clear source code. Therefore, generations of programming language designers have sought ways to segregate the handling of *exceptional conditions* that arise during execution of a program from “normal” parts of the code. The generic term for the responsible parts of a programming language is *exception handling*.

Java uses a mechanism similar to that of C++. The mechanism comprises two statements: **throw** statements signal exceptional conditions, and **try** statements do something about them.

### Syntax

*ThrowStatement:*

**throw** *Expression* ;

*TryStatement:*

**try** *Block* *Handler*<sup>+</sup> *Finally*<sub>opt</sub>

**try** *Block* *Handler*\* *Finally*

*Handler:*

**catch** ( *Parameter* ) *Block*

*Finally:*

**finally** *Block*

---

<sup>1</sup>This may partially explain why so-called *buffer overruns*, which result when a program fails to check whether it is receiving too much data, are the largest class of bug that allow security breaches in operating systems (including a weakness exploited by the infamous Internet Worm of 1988).

**Cross references:** *Expression* 119, *Block* 168, *Parameter* 100.

## 8.1 Throw Statements and Exception Types

The conventional way to indicate that something has gone wrong in a Java program, or that something “unusual” has happened, is to bundle up information about what has happened into an *exception object* and then to “throw” it. For example, you might want a certain function to check the validity of its arguments and complain if there’s something wrong:

```
/** Item #N (numbering from 0) in list L.
 * N must be >= 0, and L must be non-null. */
static List nth (List L) {
    if (N < 0 || L == null)
        throw new IllegalArgumentException ();
    ...
}
```

The operand of **throw** is just an ordinary object; this particular example uses the built-in exception type `IllegalArgumentException` (which is in package `java.lang`). The only requirement on the object thrown is that its type must be a subclass of the built-in class `Throwable` (also in `java.lang`), whose complete definition you will find in §8.4. In the example above, the exception value is created on the spot with the **new** operator, but there is no requirement that one create a new exception each time, and any exception-valued expression is fine.

The effect of **throw** is to evaluate its expression, then to complete abruptly. As usual, this causes the statement or block containing the **throw** to complete abruptly. If this process causes the body of function that contains the **throw** to complete abruptly, then the effect is to cause the *call* to that function to complete abruptly, exactly as if it was a **throw** statement with the same exception value. This process continues until either you run out of function calls (that is, the main program completes abruptly), or you reach a **try** statement (§8.2) that handles the exception.

**Implicit Throws.** The Java language defines certain legality checks that are automatically performed during execution, failure of which causes one of the built-in exceptions to be thrown implicitly. For example, the statement on the left has the same effect as that on the right:

```
x = p.value;           if (p == null)
                        throw new NullPointerException ();
                        x = p.value;
```

We summarize these implicit exception classes and other standard exception classes in Table 8.1.

## 8.2 Catching Exceptions: Try Statements

If your program makes no provision for handling exceptions, then throwing an exception basically brings your program (more accurately, the current *thread*, see §11)

to a halt. Friendly implementations generally give you a *stack trace*, which shows where (what function and line) the exception was thrown, where that function was called, and so on, thus allowing you to localize the error.

However, sometimes you anticipate getting exceptions and are prepared to do something about them. For this purpose, Java provides the **try** statement, which gives you the ability to define *handlers* for any desired set of exceptions. The generic **try** statement:

```
try {
    Normal statements
} catch (ExcpType1 e1) {
    Handler statements 1
} catch (ExcpType2 e2) {
    Handler statements 2
} catch ...
} finally {
    Clean-up statements
}
```

has the following meaning:

1. Execute the block containing the *Normal statements*.
2. If this block completes abruptly as the result of a thrown exception, let's call it *E* for reference, then check *E* against the each of the *Parameter* clauses of the handlers (if any) in turn.
  - 2a. If *E* **instanceof** *T*, where *T* is the formal parameter type of one of the handlers<sup>2</sup>, then we say that the first such matching handler *catches* the exception. Assign the exception value to the formal parameter and then execute the *Handler statements* for that handler.
  - 2b. If no handler catches the exception, then we say that the exception is *propagated*. That is, the **try** statement completes abruptly, and throws the same exception.
3. However the **try** statement completes, whether abruptly (because of a **throw** or any other kind of jump) or normally (by reaching the end of the *Normal statements*), execute the *Clean-up statements* before continuing. These statements may themselves complete abruptly, in which case the original cause of completion is "forgotten." Otherwise, if the clean-up statements complete normally, then whatever caused the completion of the **try** block is re-instated: if completed normally, then the whole **try** completes normally; if completed by a **break**, then we continue as if the clean-up statements performed that **break**; likewise for **continue**, throwing an exception, or returning a value.

---

<sup>2</sup>See §6.6.3 concerning the **instanceof** operator

It is perfectly all right for a handler to complete abruptly by means of a **throw**. Because such a **throw** does not occur within the *Normal statements* of the **try**, it causes the **try** statement itself to complete abruptly with an exception (in other words, such a **throw** is not handled by any of the handlers).

### 8.2.1 Simple example: single handler

Well, this is all extremely intricate, and some examples are definitely in order. Let's start with an example of a main program:

```
static void main (String[] args) {
    try {
        if (args.length == 1)
            print (2, Integer.parseInt (args[0]));
        else
            print (Integer.parseInt (args[0]),
                  Integer.parseInt (args[1]));
    } catch (NumberFormatException e) {
        System.err.println ("Error: improper number.");
        Usage ();
    }
}
```

The `parseInt` routine checks to make sure that its argument has the form of a valid decimal number (a string of decimal digits). If it is not, then `parseInt` throws the exception `NumberFormatException`. In this case, we have a single handler that prints an error message and then calls a function `Usage` (which is supposed to print a helpful summary of how to use this program). You can't see the actual **throw** command here; it is buried somewhere inside the implementation of `parseInt`. Nevertheless, when it is executed, that call to `parseInt` completes abruptly, as does the statement containing that call (the call to `print`), as does the **if** statement containing both of them. If both calls to `parseInt` complete normally, on the other hand, then the error message is not printed and `Usage` is not called; we simply proceed to the end of the **try** statement (which also happens to be the end of `main` in this case).

This example assumes that there are either one or two elements in the array `args`. However, it is actually possible for `args` to have no elements. In that case, attempting to evaluate `args[0]` causes the Java program to implicitly throw `ArrayIndexOutOfBoundsException`, because there is no element #0 in the array. The single handler in `main` does not handle that type of exception, so attempting this array access will cause the **try** statement to complete abruptly, as will the call to `main` (and if `main` is being used as the main program, as is usual, rather than an ordinary function, then the entire program would then terminate with an *unhandled exception*).

### 8.2.2 Multiple handlers.

Sometimes we want to respond to different exceptions in different ways, as in this example:

```
/** The first line of the file named NAME. Returns null
 * if there is no such file, or there is an error reading
 * it. Prints a warning in these cases. */
String readFirstLine (String name) {
    try {
        BufferedReader inp =
            new BufferedReader (new FileReader (name));
        String result = inp.readLine ();
        inp.close ();
        return result;
    } catch (FileNotFoundException e) {
        System.err.println ("Warning: " + name
                            + " not found.");

        return null;
    } catch (IOException e) {
        System.err.println ("Warning: I/O error on "
                            + name);

        return null;
    }
}
```

The constructor for class `FileReader` will throw `FileNotFoundException` if (as you might guess), it can't find a file with the specified name. The functions `readLine` and `close` will throw `IOException`s if something goes wrong while reading the file<sup>3</sup>. The two handlers distinguish these cases and give distinct error messages for them.

### 8.2.3 Using the Caught Exception.

In the preceding example, the order of the handlers is important: class `FileNotFoundException` is a subclass (extends) class `IOException`. Therefore, had I put the `IOException` handler first, I would never have seen the “file not found” case. In fact, we can always make do with a single handler, like this:

```
...
return result;
} catch (IOException e) {
    if (e instanceof FileNotFoundException)
        System.err.println ("Warning: " + name
                            + " not found.");
}
```

---

<sup>3</sup>This covers a lot of ground in the case of files. For example, if the file resides on another machine, that machine can crash in the middle of an operation. They're not called “exceptional” for nothing.

```

        else
            System.err.println ("Warning: I/O error on "
                               + name);
        return null;
    }
}

```

This is also the first example in which I have actually used the argument to the handler. We can, in fact, make better use of it:

```

        ...
        return result;
    } catch (IOException e) {
        System.err.println ("Warning: " + e);
        return null;
    }
}

```

This code simply converts the exception value `e` into a `String`, implicitly using the `toString` function. As it happens, class `Throwable` overrides this function to yield a somewhat more helpful message. See §8.4 for a few more details.

**Re-throwing exceptions.** Suppose that instead of returning `null` in the examples above, we instead wanted to print a warning, and then throw the same exception that had been thrown to us. Here’s the typical pattern:

```

        ...
        return result;
    } catch (IOException e) {
        System.err.println ("Warning: readFirstLine failed.");
        throw e;
    }
}

```

Since the **throw** is outside the normal statements of the enclosing **try** statement, it doesn’t get re-caught by the handler, and instead abruptly completes the **try**. This example also serves to illustrate that exceptions needn’t be created at the time they are thrown. Actually, we also have to change the function header in this example, as explained in the next section.

#### 8.2.4 The Finally Clause

It is sometimes important to take some “clean-up” action before leaving a function (or a section of a function), no matter how you leave. For example, in our `readFirstLine` example, it is important to close the stream before leaving, since otherwise the program “leaks open files,” which are generally limited resources. The **finally** clause on a **try** statement is intended for this purpose. We could re-code the example like this:

```
String readFirstLine (String name) throws IOException {
    try {
        BufferedReader inp =
            new BufferedReader (new FileReader (name));
        String result = inp.readLine ();
        return result;
    } catch (IOException e) {
        System.err.println ("Warning: " + e);
        throw e;
    }
    finally {
        inp.close ();
    }
}
```

Either we leave the *Normal statements* by means of **return**, or by re-throwing an **IOException** from the **catch** clause, or perhaps by throwing some other (unchecked) exception. In all cases, we “interrupt” this completion process to execute **inp.close()** first. Then the program returns a value or throws an exception. We are guaranteed that the program performs the **close** operation. Now, the **close** operation can itself throw an exception, in which case that takes precedence and the original event that brought us to the **finally** clause is forgotten.

### 8.3 Declaring Thrown Exceptions

Let’s step back from language details for just a moment and consider proper software design and documentation. Programs can be huge beasts, and our main tool for getting a mental grip on them is to divide and conquer—to break the larger problem ultimately into small, well-defined components. This is not sufficient in itself; after all, programs already consist of small components: individual statements. The separation into components is not effective unless each component simplifies the collection of code that constitute it by presenting a usefully condensed *interface* to the rest of the system (in the English sense of the term as opposed to the Java construct).

One intention behind the design of Java’s **class** and **interface** constructs was to provide a way of presenting such interfaces. The components, in this view, are objects and their interfaces consist of two parts:

- The *syntactic specification* of the accessible members. This consists of all the Java code of class except for the bodies of the functions. For functions, therefore, it includes of the function name, return type, and argument types.
- The *semantic specification* of the accessible members. This consists of the comments (or other documentation) on the accessible members and for the class or interface itself.

The syntactic specification tells you what the compiler will accept as a legal call of a method (or legal use of a field). The semantic specification ideally tells you what this method (or field) does and tells what additional conditions your program must fulfill to call it.

Part of this specification consists of indicating what you (the designer of the class or interface) consider to be a legal use of the methods in it. This often takes the form of *preconditions*, as in the second sentence of the following comment:

```
/** The square root of X. X must be non-negative. */
double sqrt (double x) { ... }
```

With this specification, if the programmer using `sqrt` fails to fulfill the precondition, all bets are off; the “contract” with the square-root function is broken, and the latter is free to do anything, such as to call `System.exit` and stop the program.

It is much friendlier to throw an exception instead, but if you do so, then in effect the exception becomes part of what the function “returns,” and therefore ought to be part of the syntactic specification. Java provides for exactly this sort of specification, using the **throws** clause, which was mentioned (but not defined) in §5.8.

### Syntax.

*Throws:* **throws** *ClassType*<sup>+</sup>,

This clause, which is optional in function definitions, lists a set of exceptions that may get thrown out of a given function. More accurately, it lists a set of exception types that a thrown exception might be an instance of. Seeing this clause, a user of the class knows that a call to this function may terminate abruptly with one of these exceptions—that any of these exceptions might go unhandled within the function. The clause does not mean that it actually is possible for the exception to be thrown; for example, the following is legal:

```
void check () throws IllegalStateException, IOException
{ }
```

The body is empty, so of course no exception will be thrown. Nevertheless, the **throws** clause is valid, and programmers using `check` should program as if it might throw either of the listed exceptions (by listing them, the implementor of `check` reserves the right to put something in the body that causes these exceptions).

In fact, a **throws** clause listing is mandatory whenever the body of a function can throw a *checked exception*, and it must cover all checked exceptions that might be thrown. Exceptions are all ultimately extensions of the class `Throwable`, but the set of exception classes has a bit more structure than that. Two of the built-in subclasses of `Throwable`: `Error` and `RuntimeException` (both in package `java.lang`), and any exception class that is subclass of either of them, are *unchecked exceptions*, and need not be reported in **throws** clauses (although they can be). All other exception classes are checked. For example, since the last version of `readFirstLine` in



§8.2 can throw the exception `IOException`, a checked exception, the full definition of the function must read like this:

```
String readFirstLine (String name) throws IOException {
    try {
        BufferedReader inp =
            new BufferedReader (new FileReader (name));
        String result = inp.readLine ();
        inp.close ();
        return result;
    } catch (IOException e) {
        System.err.println ("Warning: readFirstLine failed.");
        throw e;
    }
}
```

For the purposes of this rule, a function body is deemed to throw a checked exception if

1. It contains a **throw** of an exception whose static type is checked,
2. Or it contains a call to function whose **throws** clause contains a checked exception,
3. And (in either cases (2) or (3)), no **catch** clause is positioned to intercept that exception before it propagates out of the function.

Thus, the programmer is forced either to deal with each checked exception or announce in the interface that it might be propagated.

Of course, you *can* get lazy, and simply declare every function in your program like this:

```
void anyFunction (...) throws Throwable {
    ...
}
```

Then you never have bother with **try** statements. Of course, your program will terminate at the first exception, but presumably you don't care. I hope it is no surprise that I do not recommend this practice for real programs.

The design rationale behind having a category of unchecked exceptions is that they are intended for situations where the opportunities for throwing a certain exception are so pervasive that it would obscure programs to continually have to mention them. All the implicitly thrown exceptions are unchecked. This makes a good deal of sense, in general. It is difficult to turn around in Java without doing something an object, which would mean having to mention `NullPointerException` in every single function is that exception weren't unchecked. Other unchecked exceptions, such as `IllegalArgumentException` represent pervasive situations; every programmer knows that it is wrong to violate the preconditions of a function, which

is what this exception is intended to represent. You are free to add new unchecked exceptions (by extending `Error` or `RuntimeException`), but in general you should reserve such exceptions for similarly pervasive situations.

## 8.4 Exceptions in the `java.lang` Package

The Java standard libraries just bristle with exception definitions, but those defined in `java.lang` are particularly important, since they are often woven into the very fabric of the language. Table 8.1 shows the hierarchy of exceptions in `java.lang`.

It all starts with the class `Throwable`. By definition, the types of everything given to **throw** and all types mentioned in the parameters of **catch** clauses and in **throws** clauses must be subclasses of this type. Its basic definition is as follows (with all bodies replaced by semicolons):

```
public class Throwable {
    public Throwable();
    public Throwable(String message);
    public String getMessage();
    public String getLocalizedMessage();
    public void printStackTrace();
    public void printStackTrace(PrintStream s);
    public void printStackTrace(PrintWriter s);
    public Throwable fillInStackTrace();
    public String toString(); // Overrides Object.toString
}
```

Since `Throwable` defines these methods, they are all available in every exception class. Furthermore, by convention, almost all exceptions (probably including those you define) simply define constructors to parallel those of `Throwable`, and inherit everything else. Here are some more details of the methods:

**getMessage()** Every `Throwable` contains a message (which may be null). This returns that message. See the constructors for how to set the message.

**printStackTrace(...)** When you create an exception object (with **new**), Java stores in it a record of exactly where in the program this creation took place. The `printStackTrace` functions print it, either to an explicit stream or writer, if given, or (for the parameterless version) to the standard error output stream, `System.err`. The exact form of a stack trace depends on the implementation of Java you are using. At the moment, Sun's system produces something like this:

```
java.lang.NullPointerException
at glorp.mogrify(glorp.java:8)
at glorp.print(glorp.java:4)
at glorp.main(glorp.java:13)
```

Saying that (at line number 13) `main` has called `print`, which (at line number 4) has called `mogrify`, which has thrown (probably implicitly) a `NullPointerException` at line number 8.

**fillInStackTrace()** Revise the stack trace to start from the point of this call, rather than from the point of creation, and return the same exception. Sometimes, you want to revise the stack trace contained in an exception. For example, the handler

```
catch (IOException excp) {
    if (tooHardForMe (excp))
        throw (IOException) excp.fillInStackTrace ();
    ...
}
```

will re-throw `excp` if it's too hard to handle, but will tell the rest of the program that it was this handler that caused the problem (perhaps to hide the gory details).

**getLocalizedMessage()** The `Throwable` class establishes a default definition for this, which is simply to return the same value as `getMessage`. If you define new exceptions, you can override the definition with something that produces a message tailored to the “locale” in which the program is running.

**toString()** An overriding of the standard `toString` definition in `Object` that produces a string containing the name of the exception and (if non-null) the value of `getMessage`.

**Throwable(), Throwable(S)** Constructors for new exception objects either with a null message (first form) or with the given string as their message. The stack trace is filled in with the place that called the constructor.

When you create new exception classes, you will usually just follow the pattern shown here for the built-in type `Exception`: The class `Exception` is representative:

```
public class Exception extends Throwable {
    public Exception () {
        super ();
    }
    public Exception (String message) {
        super (message);
    }
}
```

That is, it looks just like `Throwable`, but with new constructors that simply call the analogous constructors in `Throwable`. Most exceptions you define will extend `Exception`. You can also extend `RuntimeException` or `Error` to get new unchecked exceptions (something you should do sparingly).

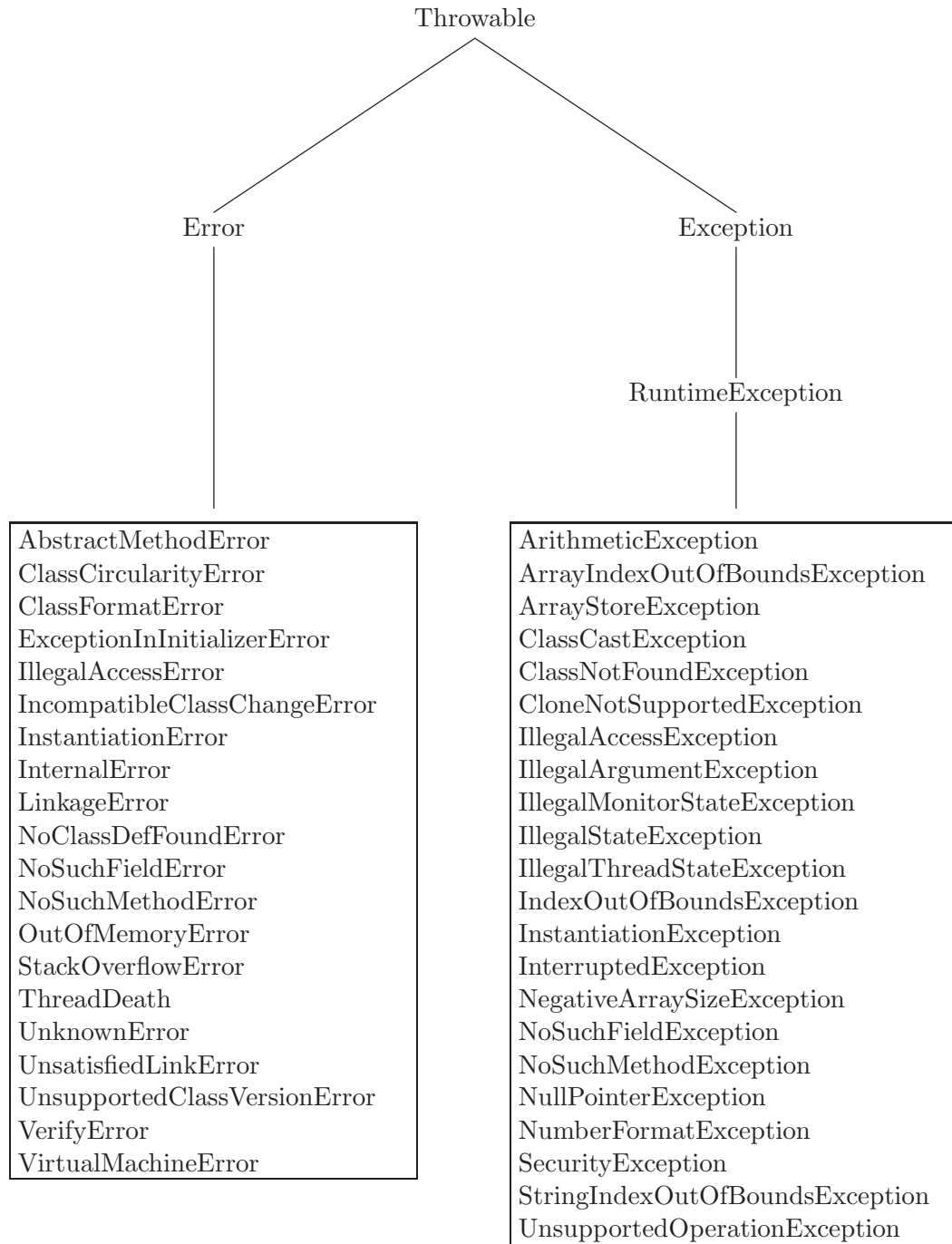


Table 8.1: The hierarchy of exceptions in `java.lang` (lines down indicate subclasses; `Throwable` is a superclass of all exceptions).

## Chapter 9

# Strings, Streams, and Patterns

In olden times (that is, until the early 1980's), essentially all communication with ordinary application programs was in the form of *text*: that is, strings (sequences) of characters, sometimes presented as broken into lines and pages. The situation is much more varied these days—what with bit-mapped displays, point-and-click, voice communication, multi-media, and the like—and, of course, computer-controlled machinery (industrial robots, on-board flight-control systems) is an exception that has been with us nearly since the beginning. Nevertheless, the manipulation of text still has an important place, and your author's somewhat antique position is that text is often a distinct improvement over some more “modern” interfaces that have displaced it.

Java provides a number of library classes that deal with plain textual data. The primitive type `char` represents single characters in the Unicode character set. Most commonly, these crop up in your programs when you extract them from `Strings`, using the `charAt` method. Corresponding to this primitive type, the Java library contains the wrapper class `Character`, which defines a number of useful functions for manipulating individual characters. The class you're likely to deal with most is `String`, which, as we've seen, gives us sequences of characters, suitable for reading from a terminal or file or for printing. This class allows you to compare strings and extract pieces of them, and thus to *parse* written notation (that is, to resolve it into its grammatical constituents and determine its meaning). However, complex analysis of text is quite tedious with the primitive tools provided by `String` and recent versions of Java have added a new class, `Pattern` (in package `java.util.regex`), which as its name suggests, provides a language for describing sets of `Strings`. Together with another new type, `java.util.Scanner` (§9.6), you can describe the format of textual input. On the other side of the coin, the construction of `Strings` generally involves the concatenation of individual pieces, some of which are constants and some of which are created dynamically. Here, too, Java 2, version 1.5 has introduced a set of convenient formatting methods, adapted from C, for complex `String` construction (§9.2.1).

Input and output by a program requires additional abstractions (§9.4). Again, these involve sequences of characters or other symbols (such as bits or bytes). However, there is a very strong bias towards producing output and consuming input

*incrementally*, so that the program need never have the complete sequence of inputs or outputs on hand simultaneously. Historically, this was for reasons of efficiency and capacity (if you are processing billions of characters using a computer with mere kilobytes of memory, you have little choice).

## 9.1 Bytes and Characters

It is often said that the business of computers is “processing symbols.” Ultimately, this means that the atomic constituents of a computer’s data come from some *finite alphabet* and these constituents’ only characteristic is that they are all distinct from one another. So, you often hear that “computers deal with 1’s and 0’s”—that the basic alphabet consists of the set of bits:  $\{0, 1\}$ . While this is true, our programs usually deal with data in somewhat larger pieces—clumps of bits, if you will. These days, the smallest of these is typically the *byte*, 8 bits, represented directly in Java as the type **byte**.

However, the fact our basic alphabet consists of numbers is misleading. The whole purpose of symbols is to *stand for* things. In the case of computers, we use numbers to stand for everything. In particular, we use them to stand for printed characters. The Java type **char**, which we think of as being a set of printed characters, is an integer type, whose members can be added and multiplied. If we want to use **chars** to represent characters (as we usually do), this becomes evident by *how we use them*. So, if the computer sends the number 113 to a printing device in a particular way, it will cause the lower-case letter ‘q’ to be printed. Thus, we humans will think of the computer as dealing with letters, whereas internally it is dealing with numbers.

### 9.1.1 ASCII and Unicode

Most programs today are written using an alphabet (or, as we computer people often say, *character set*) that goes by the name *ASCII*, standing for *American Standard Code for Information Interchange*. The same standard character set suffices for most of the textual input and output by programs. This standard defines both a set of characters and a computer-friendly encoding that maps each character to an integer in the range 0–127, as detailed in Table 9.1. This set contains ordinary, printable characters (upper- and lower-case letters, digits, punctuation marks, and blanks), and an assortment of non-printing *control characters* (so called because in the old days, they used to be used to control teletypes or other communication devices). Among the control characters, there are several *format effectors* whose effect is to control spacing and line breaks—things like horizontal and vertical tabs, line feed, carriage return, form feed (which skips to a new page), and backspace. See §6.3.1 for a further discussion.

Needless to say, there are both technical and political consequences to having a character set that is called “American” and that only has room for 127 characters. To accommodate an international clientele, a group called the Unicode Consortium has developed a much more extensive character set (generally called *Unicode*) with

65536 ( $2^{16}$ ) possible codes that incorporates many of the world's alphabets. Its first 127 characters are a copy of ASCII (so a program written entirely in ASCII is also written in Unicode). The other characters may appear in Java programs, just like ordinary ASCII characters. That is, those that are letters or digits (in any alphabet) are perfectly legal in identifiers, and all the extra characters are legal as character or string literals (§9.2, S6.3.1) and in comments.

Unfortunately, most of our compilers, editors, and so forth are geared to the ASCII character set. Therefore, Java defines a convention whereby any Unicode character may be written with ASCII characters. Before they do anything else to your program, Java translators will convert notation `\uabcd` (where the four italic letters are hexadecimal digits) into the Unicode character whose encoding is  $abcd_{16}$ . The notation actually allows you to insert any number of 'u', as in `\uuu0273`. In principle, you could write the program fragment

```
x= 3;
y=2;
```

as

```
\u0078\u003d\u0020\u0033\u003b\u000a\u0079\u003d\u0032\u003b
```

but who would want to? The intention is to use Unicode for cases where you need non-ASCII letters. For example, if you wanted to write

```
double  $\delta$  =  $\Psi_1$  -  $\Psi_0$ ;
```

you could write

```
double \u03b4 = \u03a8_1 - \u03a8_0;
```

and the compiler would be perfectly happy. Alas, getting this printed with real Greek letters is a different story, requiring the right editors and other programming tools. That topic is somewhat beyond our scope here.

Most of the early work in computers was done in the United States and Western Europe. In the United States, the ASCII character set (requiring codes 0–127) suffices for most text, and the addition of what are called the Latin-1 symbols (codes 128–255) take care of most of Europe's needs (they include characters such as 'é' and 'ö'). Therefore, characters in those parts of the world fit into 8 bits, and there has been a tendency to use the terms "byte" and "character" as if they were synonymous. This confusion remains in Java to some extent, although to a lesser extent than predecessors such as C. For example, as you will see in §9.4, files are treated as streams of bytes, but the methods we programmers apply to read or write generally deal in **chars**. There is some internal translation that the Java library performs between the two (which in the United States usually consists in throwing away half of each **char** on output and filling in half of each **char** with 0 on input) so that we don't usually have to think about the discrepancy.

$d_0$	$d_1$															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	$\text{^@}$	$\text{^A}$	$\text{^B}$	$\text{^C}$	$\text{^D}$	$\text{^E}$	$\text{^F}$	$\text{^G}$	$\text{^H}$	$\text{^I}$	$\text{^J}$	$\text{^K}$	$\text{^L}$	$\text{^M}$	$\text{^N}$	$\text{^O}$
1	$\text{^P}$	$\text{^Q}$	$\text{^R}$	$\text{^S}$	$\text{^T}$	$\text{^U}$	$\text{^V}$	$\text{^W}$	$\text{^X}$	$\text{^Y}$	$\text{^Z}$	$\text{^[}$	$\text{^\backslash}$	$\text{^]}$	$\text{^^}$	$\text{^_}$
2	$\text{^_}$	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[		]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Table 9.1: ASCII codes (equivalent to the first 128 Unicode characters). The code for the character at line  $d_0$  and column  $d_1$  is  $0\text{x}d_0d_1$ . The characters preceded by carats  $\text{^}$  are control characters, further described in Table 9.2; the notation means that the character  $\text{^}c$  can be produced on a typical computer keyboard by holding down the control-shift key while typing  $c$ .

Table of Control Characters					
Name	Ctrl	Meaning	Name	Ctrl	Meaning
NUL	$\text{^@}$	Null	DLE	$\text{^p}$	Data link escape
SOH	$\text{^a}$	Start of heading	DC1	$\text{^q}$	Device control 1
STX	$\text{^b}$	Start of text	DC2	$\text{^r}$	Device control 2
ETX	$\text{^c}$	End of text	DC3	$\text{^s}$	Device control 3
EOT	$\text{^d}$	End of transmission	DC4	$\text{^t}$	Device control 4
ENQ	$\text{^e}$	Enquiry	NAK	$\text{^u}$	Negative acknowledge
ACK	$\text{^f}$	Acknowledge	SYN	$\text{^v}$	Synchronize
BEL	$\text{^g}$	Bell	ETC	$\text{^w}$	End transmitted block
BS	$\text{^h}$	Backspace	CAN	$\text{^x}$	Cancel
HT	$\text{^i}$	Horizontal tab	EM	$\text{^y}$	End of medium
LF	$\text{^j}$	Line feed	SUB	$\text{^z}$	Substitute
VT	$\text{^k}$	Vertical tab	ESC	$\text{^[}$	Escape
FF	$\text{^l}$	Form feed	FS	$\text{^\backslash}$	File separator
CR	$\text{^m}$	Carriage return	GS	$\text{^]}$	Group separator
SO	$\text{^n}$	Shift out	RS	$\text{^^}$	Record separator
SI	$\text{^o}$	Shift in	US	$\text{^_}$	Unit separator
			DEL	$\text{^?}$	Delete or rubout

Table 9.2: ASCII control characters and their original meanings.



### 9.1.2 The class `Character`

Besides its use as a wrapper for primitive characters (see §10.5), the class `Character` is stuffed with useful methods for handling **chars**. The ones most likely to be of interest first are the ones that distinguish types of characters:

`Character.isWhitespace(c)` is true iff **char** *c* is “whitespace”—that is, a blank, tab, newline (end-of-line in Unix), carriage return, form feed (new page), or one of a few other esoteric control characters.

`Character.isDigit(c)`, `Character.isLetter(c)`, `Character.isLetterOrDigit(c)` test whether *c* is a digit (`'0'–'9'`), letter (`'a'–'z'` and `'A'–'Z'`). Actually, the complete set of letters and digits is much larger, comprising most of the world’s major alphabets.

`Character.isUpperCase(c)`, `Character.isLowerCase(c)` test for upper- and lower-case letters.

As a stylistic matter, it is always better to use these than to test “by hand:”

```
/* Good */           /* Bad */           /* REALLY Bad */
if (Character.isDigit(x))  if (x>='0' && x<='9')  if (x>=48 && x<=57)
```

A few other routines are sometimes useful for text manipulation:

`Character.toLowerCase(c)`, `Character.toUpperCase(c)` return the lower- or upper-case equivalent of *c*, if it’s a letter, and otherwise just *c*.

`Character.digit(c, r)` returns the integer equivalent of *c* as a radix-*r* digit, or `-1` if *c* isn’t such a digit. So `digit('3', 10) == 3`, `digit('A', 16) == 10`, `digit('9', 8) == -1`.

`Character.forDigit(n, r)` converts numbers back into digits. So `forDigit(5, 10) == '5'`, `forDigit(11, 16) == 'b'`.

If you happen to have a method where you use these a lot, you’ll soon get tired of writing `Character..` Fortunately, placing

```
import static java.lang.Character.*;
```

among the **import** statements at the beginning of a program file will make it unnecessary to write the qualifier.

## 9.2 Strings

Pages 203–205 list the headers of the extensive set of operations available on **Strings**. In addition, the core Java language provides some useful shorthand syntax for using some of these operations, as described in §9.2

Objects of type **String** are *immutable*: that is, the contents of the object pointed to by a particular reference value never changes. It is not possible to “set the fifth character of string *y* to *c*.” Instead, you set *y* to an entirely new string, so that after

```

y = "The bat in the hat is back.";
x = y;
y = y.substring (0, 4) + 'c' + y.substring (5);

```

the variable `y` now points to the string

```
The cat in the hat is back.
```

while `x` (containing `y`'s original value) still points to

```
The bat in the hat is back.
```

This property of `Strings` makes certain uses of them a bit clumsy or slow, so there is another class, `StringBuilder`, whose individual characters *can* be changed (see §9.3).

### 9.2.1 Constructing Strings

You will almost never have to construct `Strings` using `new String(...)`, so I have not bothered to show any in Figure 9.1. String literals (like `" "` or `"Hello!"`) handle most constant strings.

The `valueOf` methods allow you to form `Strings` from primitive values. For example,

```

String.valueOf (130+3)   is "133"
String.valueOf (true)    is "true"
String.valueOf (1.0/3.0) is "0.3333333333333333"

```

Normally, you don't need these routines, because the concatenation operator (`+`) on `String` uses `valueOf` to convert non-string arguments. For example,

```

"The answer is " + (130 + 3)   is "The answer is 133"
"" + (130 + 3)                  is "133"

```

When applied to `Objects` (reference-type values), `valueOf` (and therefore `String` concatenation) becomes particularly interesting. For non-null reference values, it uses the `toString` operator to convert the object to a `String`. Now, this function is defined in class `Object`, and therefore can be overridden in *any* user-defined type. You can therefore control how types you create are printed—a very pretty example of object-oriented function dispatching. For example, suppose I define

```

/** A point in 2-space. */
class Point {
    public double x, y;
    ...
    public String toString () {
        return "(" + x + "," + y + ")";
    }
}

```

```

package java.lang;
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    /* Unless otherwise noted, the following routines will throw a
     * NullPointerException if given a null argument and
     * IndexOutOfBoundsException if a non-existent portion of a
     * String is referenced. */

    /** A Comparator for which
     *     compare(s0, s1) == s0.compareToIgnoreCase (s1)
     * for Strings s0 and s1 (used for specifying the order of
     * Strings in sorted lists and the like). */
    public static final java.util.Comparator<String> CASE_INSENSITIVE_ORDER;

        /* Conversions and translations */
    /** The following 6 functions all return a printable
     * representation of their argument. */
    public static String valueOf(char x);
    public static String valueOf(double x);
    public static String valueOf(float x);
    public static String valueOf(int x);
    public static String valueOf(long x);
    public static String valueOf(boolean x);
    /** If OBJ == NULL, then the string "null". Otherwise,
     * same as OBJ.toString (). */
    public static String valueOf(Object obj);

    /** A String that is formed by converting the trailing arguments
     * (0 or more Objects) into Strings according to the pattern given
     * in FORMAT. See §9.2.1 for details. */
    public String format (String format, Object ... args);

        /* Accessors */
    /** The length of this. */
    public int length();
    /** Character #K of this (numbering starts at 0). */
    public char charAt(int k);
    /** The String whose characters are charAt(b)..charAt(e-1). */
    public String substring(int b, int e);
    /** Same as substring(b, length()). */
    public String substring(int b);

```

*Continues...*

Figure 9.1: Excerpts from class String, part 1. Excludes bodies.

*Class String, continued.*

```

        /* Predicates */
    /** Compare this String to STR lexicographically. Returns a
     * value less than 0 if this String compares less than STR,
     * greater than 0 if this String is greater than STR, and 0
     * if they are equal. */
    public int compareTo(String str);
    /** Same as compareTo ((String) obj). Thus, throws a
     * ClassCastException if OBJ is not a String. */
    public int compareTo(Object obj);
    /** As for compareTo(STR), but ignores the distinction between
     * upper- and lower-case letters. */
    public int compareToIgnoreCase(String str);
    /** If FOLD is false, equivalent to
     *     substring(K0,K0+LEN).compareTo(S.substring(K1,K1+LEN)) == 0,
     * and otherwise (if FOLD is true) equivalent to
     *     0 == substring(K0, K0+LEN)
     *         .compareToIgnoreCase (S.substring(K1, K1+LEN))
     * except that it returns false if either substring operation
     * would cause an exception. */
    public boolean regionMatches(boolean fold, int k0,
                               String S, int k1, int len);
    /** Same as regionMatches (false, K0, S, K1, LEN) */
    public boolean regionMatches(int k0, String S, int k1, int len);
    /** Same as
     *     OBJ instanceof String && compareTo ((String) OBJ) == 0. */
    public boolean equals(Object obj);
    /** Same as compareToIgnoreCase (STR) == 0. */
    public boolean equalsIgnoreCase(String str);
    /** Same as regionMatches (K0, STR, 0, STR.length ()) */
    public boolean startsWith(String str, int k0);
    /** Same as startsWith (0, STR) */
    public boolean startsWith(String str);
    /** Same as startsWith (length () - STR.length (), STR). */
    public boolean endsWith(String str);

    /* Conversions and translations */
    /** The following four routines return lower- or upper-case
     * versions of this String. The LOCALE argument, if present,
     * modifies the result in the case of Turkish. */
    public String toLowerCase();
    public String toUpperCase();

```

*Continues...*

Figure 9.1, continued: The class String, part 2.

*Class String, continued.*

```

        /* Non-destructive modifications */
    /** Returns a new string consisting of this string concatenated
     * with S. */
    public String concat(String S);
    /** A string copied from this, but with all instances of C0
     * replaced by C1. */
    public String replace(char c0, char c1);
    /** The string resulting from trimming all characters that are
     * <= ' ' (space) from both ends of this. Thus, trims blanks,
     * tabs, form-feeds, and in fact, all ASCII control
     * characters. */
    public String trim();

        /* Searches */
    /** The smallest (first) k>=S such that C == charAt(k), or -1
     * if there is no such k. */
    public int indexOf(int c, int s);
    /** Same as indexOf (C, 0). */
    public int indexOf(int c);
    /** The largest (last) k<=S such that C == charAt(k), or -1
     * if there is no such k. */
    public int lastIndexOf(int c, int s);
    /** Same as lastIndexOf (C, length ()-1). */
    public int lastIndexOf(int c);

    /** The smallest (first) k>=S such that startsWith (TARGET, S),
     * or -1 if there is none. */
    public int indexOf(String target, int s);
    /** Same as indexOf (TARGET, 0) */
    public int indexOf(String target);
    /** The largest (last) k<=S such that startsWith (TARGET, S), or
     * -1 if there is none. The last occurrence of the empty
     * string, "", is at length (). */
    public int lastIndexOf(String target, int s);
    /** Same as lastIndexOf (TARGET, length ()) */
    public int lastIndexOf(String target);

        /* Miscellaneous. */
    /** Returns this string. */
    public String toString();
    /** Returns an integer hash code for this string. The actual
     * code returned is the Java int equal to
     *  $s_0 31^{n-1} + s_1 31^{n-2} + \dots + s_{n-1} \bmod 2^{32}$  */
    public int hashCode();
    /** A String .equal to this and having the property that
     *  $s_0.equals(s_1)$  iff  $s_0.intern() == s_1.intern()$ . */
    public String intern();
}

```

Figure 9.1, continued: The class String, part 3.

Now if in a program I write something like

```
Point start = ...;
...
System.out.println ("Starting point: " + start);
// same as "Starting point: " + String.valueOf (start)
```

I could see printed

```
Starting point: (1.1,-2.3)
```

There is a default `toString` that works for all objects. On `String`, it is simply the identity function.

To construct more complicated `Strings`, you will generally use `A.concat(B)`, more often written conveniently as `A + B`, or the extremely powerful method `format`, a welcome addition to Java that came in with Java 2, version 1.5.

To understand the idea behind `format`, it's best to look at an example. Suppose that I am trying to create a `String` of the form

```
"Average of 1000 inputs: 29.65, standard deviation: 4.81."
```

where the numbers come from variables `N`, `mean`, and `sigma`. I can use concatenation and write this as

```
"Average of " + N + " inputs: " + mean + ", standard deviation: " + sigma;
```

This works, but has the slight problem that `mean` and `sigma` get printed with however many decimal places it takes to represent their value unambiguously, whereas we were content with two places. You might also object that it looks a tad messy. Now, there is another way to produce the `String` we want:

```
String.format ("Average of %d inputs: %.2f, standard deviation: %.2f",
               N, mean, sigma);
```

which inserts the values of `N`, `mean`, and `sigma` at the places indicated by the *format specifiers* `%d` (which means, “format as a decimal integer”), and `%.2f` (which means “format as a decimal numeral rounded to two places to the right of the decimal point”). C programmers will recognize the format specifiers from the `printf` series of functions in the C library. The first argument to `format` is called the *format string*, and the rest are simply additional arguments, which we index from 1 (so `N` is argument 1, and `sigma` is argument 3).

Java has an extensive set of format specifiers, including a whole set (not found in C) that handles dates and times. You'll find a full description in the on-line documentation for `java.util.Formatter`. Here, we'll just do a quick summary of some useful ones.

Most format specifiers have the general form

```
%[K][flags][W][.P]C
```

where square brackets (`'[...]'`) here indicate optional parts:

*C* specifies the *conversion* to be applied to the argument to make it into a `String`. In most cases, it is a single character.

*K* is an optional integer index, indicating which of the arguments is to be converted. By default, `format` will take the arguments in order, so that our format-string example above is equivalent to

```
"Average of %1$d inputs: %2$.2f, standard deviation: %3$.2f",
```

*W* is an optional non-negative integer width, giving the minimum size of the converted string.

*P* is an optional non-negative integer precision, that restricts the number of characters in some way, depending on *C*.

*flags* are characters that modify the meaning of *C*.

Table 9.3 shows some of the most useful values of *C*. The categories in that table refer to possible types of arguments:

**integral** the integer numeric types `int`, `short`, `long`, and `byte` (or more accurately, the corresponding “boxed” types `Integer`, `Short`, `Long`, and `Byte`: see §10.5), plus `java.math.BigInteger` (a class that represents all the integers, or at least those that fit in memory).

**general** any type.

**floating point** the numeric types `float` and `double` (again, more accurately `Float` and `Double`), plus `java.math.BigDecimal` (a class that represents all rational numbers expressible in finite decimal notation).

**none** doesn’t use an argument.

The flag characters generally have different meanings for different types, as detailed above. However, the ‘-’ flag always means “left justify within the specified width (*W*)”. So,

```
String.format ("|%-5d|%5d|", 13, 13) produces "|13 | 13|"
```

### 9.2.2 String Accessors

The accessors let you get at individual characters or substrings of a `String`. Characters are numbers from 0, so that the last is at index `length()-1`. Substrings are specified by giving the index of the first character of the substring and the index just beyond that of the last character, which seems a little odd, but has its advantages. An attempt to fetch non-existent characters raises the exception `IndexOutOfBoundsException`.

Conversion	Category	Description
'd'	integral	format as a decimal integer. With the '0' flag, pad on the left with 0s if necessary to get width <i>W</i> .
'o'	integral	format as an octal (base 8) integer. With the '#' flag, always start with 0 digit. The '0' flag works as for 'd'.
'x', 'X'	integral	format as a hexadecimal (base 16) integer. With 'X', use upper-case letters. The '0' flag works as for 'd'. With the '#' flag, always start with '0x' or '0X'.
'f'	floating point	format as a number with decimal point without an ( <i>fixed point</i> , as in 186000). If present, <i>P</i> gives the number of decimal places.
'e', 'E'	floating point	format in scientific notation (as in 1.86e5). If present, <i>P</i> is the number of digits after the decimal point. With 'E', uses upper-case 'E' (1.86E5).
'g', 'G'	floating point	format in 'f' or 'e' ('E' in the case of 'G') format, depending on size. If present, <i>P</i> is the number of significant figures.
's', 'S'	general	convert to <code>String</code> , usually using <code>toString</code> . With 'S', convert to upper case first. If present, <i>P</i> is the maximum number of characters (any extra characters are dropped). It's possible for a type to provide even more control over its '%s' format by implementing the <code>Formattable</code> interface. See the full documentation.
'n'	none	an end of line on output.
'%'	none	a percent sign (so, to output "33%", you'd use the format specifiers "%d%%").

Table 9.3: Common format conversion characters and their meanings. See the text for the meaning of "category."



**Example: Squeeze out selected character.**

```
/** Return S with all occurrences of C removed. */
static String squeeze (String S, char c) {
    String S2;
    S2 = "";
    for (int i = 0; i < S.length; i += 1)
        if (S.charAt (i) != c)
            S2 += S.charAt (i);
    return S2;
}
```

Here is another method, using substring:

```
/** Return S with all occurrences of C removed. */
static String squeeze (String S, char c) {
    int i;
    i = 0;
    while (i < S.length) {
        if (S.charAt (i) != c)
            i += 1;
        else
            S = S.substring (0, i) + S.substring (i+1);
    }
    return S;
}
```

The original string object is *not* changed in either of these (indeed, it is impossible to do so).

### 9.2.3 String Comparisons, Tests, and Searches

One very common pitfall (sometimes even for experienced programmer) is that the equality operator (==) does *not* test equality of the *contents* in **Strings**. When applied to **Strings**, it has its usual meaning: pointer equality. Within a given class, all character-by-character identical string literals and constant **String** expressions refer to the same **String** object; however, every application of **new** creates a new object, so that

```
String x = "Foo";
x == new String (x)           // false
new String (x) == new String (x) // false
x == x                        // true
x == "Foo"                    // true (in same class)
x.equals (new String (x))     // true
```

The **intern** function on strings in effect chooses a single **String** object amongst all those that are equal (contain the same character sequence). However, I suggest that

you generally avoid such tricks except where speed is needed, and use the `equals` and `compareTo` methods (and their ilk) for comparing strings.

### 9.2.4 String Hashing

The `hashCode` function is a hashing function that facilitates the creation of search structures for strings. Because it depends only on the characters in the string, it has the necessary property for all hash functions, that

$$S_0.\text{hashCode}() == S_1.\text{hashCode}() \text{ if } S_0.\text{equals}(S_1)$$

for strings  $S_0$  and  $S_1$ . You might think that the function described in the documentation comment is slow to compute, but because  $31 = 2^5 - 1$ , the following fairly inexpensive function computes the same value:

```
/** Compute S.hashCode () */
public static int hashCode (String s) {
    int r;
    r = 0;
    for (int i = 0; i < s.length (); i += 1)
        r = (r<<5) - r + s.charAt (i);
    return r;
}
```

## 9.3 StringBuilders

You can think of a `StringBuilder` as a *mutable* (modifiable) `String`<sup>1</sup>. Use them to build up strings quickly (that is, without having to create a new object each time you add a character). In fact, the Java system itself uses `StringBuilders` behind the scenes to implement the '+' operator on `Strings`.

For example, suppose you wanted to take an array of integers and create a `String` consisting of those integers separated from each other by slashes ('/'). That is, you'd like

`join (new int[] { 42, -128, 70, 0 })` to produce "42/-128/70/0"

Here's one approach:

```
public static String join (int[] A) {
    StringBuilder result = new StringBuilder ();
    for (int i = 0; i < A.length; i += 1) {
        if (i > 0)
```

---

<sup>1</sup>The `StringBuilder` class is new with Java 1.5. It is essentially identical to an older class, `StringBuffer`, but its methods are significantly faster because they don't attempt to deal with situations where multiple threads try to operate on the same `StringBuilder` simultaneously (such simultaneous operations are free to blow up in arbitrary ways). This makes perfect sense, since one seldom intends to have more than one thread add text to any given `StringBuilder`.

```
        result.append ('/');
        result.append (A[i]);
    }
    return result.toString ();
}
```

## 9.4 Readers, Writers, Streams, and Files

At some point, useful programs communicate with the outside world. While graphical user interfaces can be useful for interaction, there is a great deal of communication that can be described simply as the transmission of a sequence of characters, bytes, bits, or generally symbols from some finite alphabet. When your browser requests a Web page, for example, what actually happens under the hood is that it codes its request as a stream of characters and receives in return a stream of characters, which it then interprets as fancily formatted page. To the programs that manipulate them, everything from Java source programs to Microsoft Word documents to error messages written on your screen look like streams of characters. So it's not surprising to find that the Java library contains numerous classes and interfaces dedicated to several forms of the abstract notion of "a stream of symbols."

This area is an instance of where object-oriented programming languages can be useful. The problem is that there are many potential sources and destinations for streams of characters, but the programs that deal with those characters (interpret or produce them) are indifferent to their source, and might be useful for streams of many different origins. As with other situations of this sort, we react by defining an abstraction that captures just those universal characteristics of a "stream of characters" that programs need and expressing this abstraction within our programming language as some kind of interface.

An obvious question to ask is why a "stream of characters" should be anything other than an "array of characters" or "list of characters." What justification is there for a new concept? There are a couple of reasons:

- There is not necessarily a way to know the *length* of a stream of characters. Indeed, it need not have any finite length *a priori*.
- It is the nature of an array or list that one always has access to its entirety. There is no notion of "finishing with" element *k* of a list; if you accessed it once, your program is perfectly entitled to access it again. The implementation is required to keep all data in the an array or list available, so that the size of a program becomes proportional to the amount of data it processes. Needless to say, this is a problem in the case of input to a program that runs indefinitely. However, it is also an efficiency issue for programs that access data *sequentially*, and don't need to go back to element *k* after they've finished with it.

In Java, the situation is rather complicated, because the language and its library have been through a number of major revisions, each one of which seems to have

```

package java.lang;
public final class StringBuilder
    implements java.io.Serializable, CharSequence {
    /* Unless otherwise noted, the following routines will throw a
     * NullPointerException if given a null argument and
     * IndexOutOfBoundsException if a non-existent portion of a String
     * or StringBuilder is referenced. */

    /* Constructors */
    /** An empty (length() == 0) StringBuilder. */
    public StringBuilder();
    /** A new StringBuilder whose initial contents are INIT */
    public StringBuilder(String init);

    /* Accessors */
    /** The current length of this. */
    public int length();
    /** Character #K of this (numbering starts at 0). */
    public char charAt(int k);
    /** The String whose characters are charAt(b)..charAt(e-1). */
    public String substring(int b, int e);
    /** Same as substring(b, length()). */
    public String substring(int b);

    /** Sets DEST[D0] to charAt(B0), DEST[D0+1] to charAt(B0+1), ...
     * up but not including charAt(ENO0). Exception if DEST is
     * null or not big enough. */
    public void getChars(int b0, int end0, char[] dest, int d0);

    /** The contents of this as a String. (Further changes to this
     * StringBuilder don't affect the String.) */
    public String toString();

```

*Continues...*

Figure 9.2: The class `StringBuilder`, part 1. Excludes bodies and deprecated (obsolete) functions.

```

        /* Modifiers */
    /** Insert X into this StringBuilder, putting its first character
     *  at position K (0 <= K <= length()). Move the original
     *  characters starting at K to the right to make room, and
     *  increase the length as needed. */
    public StringBuilder insert(int k, String x);

    /** Each remaining insert operation, insert(k, ARGS), is
     *  equivalent to insert(k, String.valueOf(ARGS)). */
    public StringBuilder insert(int k, char x);
    public StringBuilder insert(int k, double x);
    public StringBuilder insert(int k, float x);
    public StringBuilder insert(int k, int x);
    public StringBuilder insert(int k, long x);
    public StringBuilder insert(int k, boolean x);
    public StringBuilder insert(int k, Object obj);
    public StringBuilder insert(int k, char[] C, int k0, int len);
    public StringBuilder insert(int k, char[] C);

    /** The operations append(ARGS) are all equivalent to
     *  insert(length(), ARGS) */
    public StringBuilder append(char x);
    public StringBuilder append(double x);
    public StringBuilder append(float x);
    public StringBuilder append(int x);
    public StringBuilder append(long x);
    public StringBuilder append(boolean x);
    public StringBuilder append(Object x);
    public StringBuilder append(String x);
    public StringBuilder append(char[] C, int k0, int len);
    public StringBuilder append(char[] C);

```

*Continues...*

Figure 9.2, continued: The class `StringBuilder`, part 2.

```
/** Remove the characters at positions START..END-1 from this
 * StringBuilder, moving the following characters over and
 * decrementing the length appropriately. Return this. */
public StringBuilder delete(int start, int end);
/** Same as delete(k, k+1) */
public StringBuilder deleteCharAt(int k);

/** Same as delete(START, END).insert(START, X) */
public StringBuilder replace(int start, int end, String x);
/** Reverse the sequence of characters in this; return this. */
public StringBuilder reverse();
/** Causes charAt(K) to be C. Exception if charAt(K) would cause
 * an exception */
public void setCharAt(int k, char c);
/** Causes length() to be LEN. If len0 is the previous length,
 * then charAt(k) is unchanged for 0 <= k < len0, and
 * charAt(k)=0 for len0 <= k < LEN. */
public void setLength(int len);
}
```

Figure 9.2, continued: The class `StringBuilder`, part 3.

taken a different approach to the problem. The result is that the current library contains *all* the solutions of its prior versions, leading to quite a bit of redundancy and confusion. We'll try to sort this out by picking and choosing our topics and concentrating on how to do useful things.

### 9.4.1 Input

The classes handling input in Java are loosely organized as follows:

- The abstract class `java.io.InputStream` represents a stream of **bytes**.
- A large number of concrete classes implement `InputStream`. In particular, these include `java.io.FileInputStream`, which produces the contents of a file as a stream of bytes. The *standard input* (which by default is the stream of data you type at the terminal) is presented to your program as an `InputStream` (called `System.in`), although the language is silent as to which concrete implementation it is.
- The abstract class `java.io.Reader` represents a stream of **chars**.
- There are large number of concrete implementations of `Reader`, including:

`java.io.InputStreamReader` takes an `InputStream` and presents it as a `Reader`.

`java.io.StringReader` takes a `String` and presents its characters as a `Reader`.

`java.io.BufferedReader` is an implementation of `Reader` that takes any kind of `Reader` and makes it into a more efficient one by reading large segments at a time.

`java.util.Scanner` is a class that takes a `Reader` or an `InputStream` and delivers its characters clumped into useful *tokens* (see §9.6).

For example, a program that reads words from the standard input might set things up as follows:

```
import java.io.*;
import java.util.Scanner;

class WordReader {
    public static void main (String[] args) {
        Scanner input = new Scanner (System.in);
        processWords (input);
    }
    ...
}
```

after which, the `next` method in `input` will deliver words from the input, one at a time, as described in §9.6.

Sometimes, you'll instead want to fetch the input from a named file. For example, you might want to run your program like this:

```
java DoMyTaxes tax-2004.dat
```

and have it mean that your program is to read its input from the file named `tax-2004.dat`. To get this effect, you could write:

```
import java.io.*;
import java.util.Scanner;

public class DoMyTaxes {
    public static void main (String[] args) {
        try {
            Scanner input = new Scanner (new FileInputStream (args[0]));
            processTaxes (input);
        } catch (FileNotFoundException e) {
            System.err.println ("Could not open file " + args[0]);
            System.exit (1);
        }
    }
    ...
}
```

Many programs allow you to take input either way, so that plain

```
java DoMyTaxes
```

reads input from the terminal. For this, we put the two cases above together:

```
import java.io.*;
import java.util.Scanner;

public class DoMyTaxes {
    public static void main (String[] args) {
        try {
            Scanner input;
            if (args.length == 0)
                input = new Scanner (System.in);
            else
                input = new Scanner (new FileInputStream (args[0]));
            processTaxes (input);
        } catch (FileNotFoundException e) {
            System.err.println ("Could not open file " + args[0]);
            System.exit (1);
        }
    }
    ...
}
```

Finally, some programs allow you the option of putting the input directly onto the command line, so that `args[0]` isn't just the name of the input; it *is* the input:



```

import java.io.*;
import java.util.Scanner;

public class DoCommands {
    public static void main (String[] args) {
        Scanner input = new Scanner (new StringReader (args[0]));
        processCommands (input);
    }
    ...

```

### 9.4.2 Readers and InputStreams

The classes `Reader` and `InputStream` are abstract. They do not, in and of themselves, define any particular source of input. In the examples above, think of `FileInputStream` and `StringReader` as types of vacuum cleaners. A `Scanner` is an attachment that one can place on either of these. The types `Reader` and `InputStream` are like standard nozzle shapes: you can attach a `Scanner` to any kind of nozzle that conforms to `Reader` or `InputStream`.

Thus, `InputStream` describes a family of classes that support the following methods (among others):

`read()` Return the next byte of input (as an **int** in the range 0–255) or -1 if at the end.

`read(b, k, len)` where  $b$  is an array of **bytes**, reads at most  $len$  bytes into  $b[k]$ ,  $b[k+1]$ , ... and returns the number read. This method is here for efficiency; it is often faster to read many bytes at once.

`close()` shuts down the stream. Such an operation is often necessary to tell the system when input is no longer needed, since it may otherwise have to consume time or space keeping input available.

`mark(n)` and `reset()` respectively make a note of the program's current place in the input and "rewind" the input to the last mark (assuming that no more than  $n$  characters have been read since). Depending on the kind of `InputStream`, they do not always work. So...

`markSupported()` is true iff an `InputStream` allows marking.

The `Reader` class is almost the same, but deals in **chars** rather than **bytes**.

It might occur to you that it would probably be rather clumsy to have to write your program using just `read`. Quite true: you will normally attach something more useful (typically a `Scanner` in this class) to your `Readers` or `InputStreams`, rather than using them directly. On the other hand, you do have to be familiar with these classes if you wish to create a new kind of input source and want it to be "plug-to-plug compatible" with existing classes that attach to `Readers` and `InputStreams`.

### 9.4.3 Output

The classes handling output in Java are roughly analogous to those for input, but with data going in the “opposite direction.”

- The abstract class `java.io.OutputStream` absorbs a stream of **bytes**.
- Numerous concrete classes implement `OutputStream`, notably:

`java.io.FileOutputStream` writes its stream of bytes into a file.

`java.io.BufferedOutputStream` collects its stream of bytes and feeds it to another `OutputStream` in chunks, for efficiency.

**`java.io.PrintStream`** adds methods to `OutputStream` that make it easy to write things such as strings and numbers in their printed forms. The standard output and standard error output streams of a program are both `PrintStreams`. Generally, `PrintStreams` are set up to send their streams of bytes to another `OutputStream` (such as a `FileOutputStream`).

- The abstract class `java.io.Writer` absorbs a stream of **chars**.
- Concrete implementations of `Writer` include:

`java.io.OutputStreamWriter` converts its stream of **chars** that it receives into a stream of bytes and gives it to an `OutputStream`.

`java.io.StringWriter` collects its stream of **chars** into a `String` that you can extract at any time with `toString`.

`java.io.BufferedWriter` is analogous to `BufferedOutputStream`, but connects to another `Writer` and deals in **chars**.

`java.io.PrintWriter` is analogous to `PrintStream`.

### 9.4.4 PrintStreams and PrintWriters

Especially when creating output intended for human consumption, writing things one character at a time is rather clumsy. The `PrintStream` and `PrintWriter` classes in `java.io` are intended to make this more convenient. Both of these have a set of `print` and `println` methods that convert their argument to a `String` and then write its characters. In `println` method, furthermore, additionally appends an end-of-line sequence, so that the next output starts on a new line. Together with `String` concatenation (`+`), they allow you to write just about anything. For example:

```
System.out.println ("Average of " + N + " inputs: " + mean
                    + ", standard deviation: " + sigma);
```

With Java 2, version 1.5, you have the alternative of using the powerful `format` methods as well:

```
System.out.format ("Average of %d inputs: %.2f, standard deviation: %.2f%n",
                  N, mean, sigma);
```

which has the same effect as

```
System.out.print
  (String.format ("Average of %d inputs: %.2f, standard deviation: %.2f%n",
                  N, mean, sigma));
```

## 9.5 Regular Expressions and the Pattern Class

So far, all the means we've seen for breaking up a string or other stream of characters have been quite primitive, which can lead to some rather involved programs to do simple things. Consider the problem of interpreting the command-line arguments to a program. A certain program might allow the following parameters

```
--output-file=FILE      --output=FILE
--verbose
--charset=unicode        --charset=ascii
--mode=N
```

where *FILE* is any non-empty sequence of characters and *N* is an integer. That is, a user might start this program with commands like

```
java MyProgram --output-file=result.txt --charset=unicode
java MyProgram --verbose
```

Suppose we want to write a function to test whether a given string has one of these formats. Here's a brute-force approach:

```
/** Return true iff OPTION is a valid command argument. */
public static boolean validArgument (String arg) {
    return (arg.startsWith ("--output-file=") && arg.length () > 14)
        || (arg.startsWith ("--output=") && arg.length () > 9)
        || arg.equals ("--verbose")
        || arg.equals ("--charset=unicode")
        || arg.equals ("--charset=ascii")
        || (arg.startsWith ("--mode=") && isInteger (arg));
}

static boolean isInteger (String numeral) {
    for (int i = 0; i < numeral.length (); i += 1)
        if (! Character.isDigit (numeral.charAt (i)))
            return false;
    return true;
}
```

Clearly, it would be nice not to have to write so much. In fact, we can write the body of `validArgument` much more succinctly like this:

```
final static String argumentPattern =
    "--output(-file)?=.+|--verbose|--charset=(unicode|ascii)"
    + "|--mode=\\d+";

public static boolean validArgument (String arg) {
    return (arg.matches (argumentPattern));
}
```

(Here, we used a separate constant declaration for the pattern string and used ‘+’ to break it across lines for better readability. We could have dispensed with the definition of `argumentPattern` and simply written the pattern string into the `return` statement.)

Here, `matches` method on `Strings` treats `argumentPattern` as a *regular expression*, which is a description of a set of strings. To *match* a regular expression is to be a member of this set.

In the package `java.util.regex`, Java provides classes `Pattern` and `Matcher`, which provide, respectively, a highly augmented version of regular expression, and a kind of engine for matching strings to them. Although the term “regular expression” is often used to describe things like `Pattern`, it is misleading. Formal language theory introduced regular expressions long ago, in a form that is much more restricted than Java’s. Since the term has been appropriated, we’ll use words like “pattern” (or “Pattern”), or “regex” instead. For a complete description of Java patterns, see the documentation for the classes `Pattern` and `Matcher`. For now, we’ll just look at some of the most common constructs, summarized in Table 9.4.

The most basic regular expressions contain no special characters, and denote simply a literal match. For example, the two expressions

```
s.equals ("--verbose")      s.matches ("--verbose")
```

have the same value. In set language, we say that the regular expression `--verbose` denotes the set containing the single string `--verbose`.

Other forms of regular expression exist to create sets that contain more than one string. For example, to see if a one-character string, `s`, consists of a single vowel (other than ‘y’), you could write either

```
s.matches ("a|e|i|o|u")      or      s.matches ("[aeiou]")
```

while to check for a (lower-case) consonant, you’ll need

```
s.matches ("[b-df-hj-np-tv-z]")
```

To test for a word starting with single consonant followed by one or more vowels, you would write

```
s.matches ("[b-df-hj-np-tv-z][aeiou]+")
```

Regular Expression	Matches
$x$	$x$ , if $x$ is any character not described below
$[c_1c_2\cdots]$	Any one of the characters $c_1, c_2, \dots$ . Any of the $c_i$ may also have the form $x-y$ , meaning “any single character $\geq x$ and $\leq y$ .”
$[\sim c_1c_2\cdots]$	Any single character <i>other than</i> $c_1, c_2, \dots$ . Here, too, you can use $x-y$ as short for all characters $\geq x$ and $\leq y$ .
$.$	Any character except (normally) line terminators (newline characters or other character sequences that mark the end of a line).
$\backslash s$	Any whitespace character (blank, tab, newline, etc.).
$\backslash S$	Any non-whitespace character.
$\backslash d$	Any digit.
$\wedge$	Matches zero characters, but only at the beginning of a line or string.
$\$$	Matches zero characters, but only at the end of a line or string.
$(R)$	Matches the same thing as $R$ . Parentheses serve as grouping operators, just as they do in arithmetic expressions.
$R^*$	Any string that can be broken into zero or more pieces, each of which matches $R$ . This is called the <i>closure</i> of $R$ .
$R^+$	Same as $RR^*$ . That is, it matches any string that can be broken into one or more pieces, each of which matches $R$ .
$R_0R_1$	Any string that consists of something that matches $R_0$ immediately followed by something that matches $R_1$ .
$R_0 R_1$	Any string that matches <i>either</i> $R_0$ or $R_1$ .

Table 9.4: Common regular expressions. Lower-case italic letters denote single characters.  $R, R_0, R_1$ , etc. are regular expressions (the definitions are recursive). As with arithmetic expressions, operators have precedences, and are listed here in decreasing order of precedence. Thus, ‘ $ab|xy^*$ ’ matches the same strings as ‘ $(ab)|(x(y^*))$ ’ rather than ‘ $a(b|x)(y^*)$ ’ or ‘ $((ab)|(xy))^*$ ’.

Finally, to check that `s` consists of a series of zero or more such words, each one followed by any amount of whitespace, you could write

```
s.matches ("([b-df-hj-np-tv-z][aeiou]+\\s+)*")
```

Here, I had to write two backslashes to create a ‘`\s`’ because in Java string literals, a single backslash is written as a double backslash.

So far, I have been writing strings and calling them regular expressions. The whole truth is somewhat more complicated. The Java library defines two classes: `Pattern` and `Matcher`, both of which are in the package `java.util.regex`. The relationship is suggested in the following program fragment:

```
/* String representing a regular expression */
String R = "[b-df-hj-np-tv-z][aeiou]+";
String s = "To be or not to be";
/* A Pattern representing the same regular expression */
Pattern P = Pattern.compile (R);
/* An object that represents the places where P matches s or
   parts of it. */
Matcher M = P.matcher (s);
if (M.matches ())
    System.out.format ("P matches all of '%s'\n", s);
if (M.lookingAt ())
    System.out.format ("P matches the beginning of '%s'\n", s);
if (M.find ())
    /* M.group () is the piece of s that was matched by the last
       matches, lookingAt, or find on M. */
    System.out.format ("P first matches '%s' in '%s'\n", M.group (), s);
if (M.find ())
    System.out.format ("P next matches '%s' in '%s'\n", M.group (), s);
if (M.find ())
    System.out.format ("P next matches '%s' in '%s'\n", M.group (), s);
```

which, for the given value of `s`, would print:

```
P first matches 'be' in 'To be or not to be'
P next matches 'no' in 'To be or not to be'
P next matches 'to' in 'To be or not to be'
```

A `Pattern` is a “compiled” regular expression, and a `Matcher` is something that finds matches for a given `Pattern` inside a given `String`. The reason for wanting to compile a regular expression into a `Pattern` is mostly one of efficiency: it takes time to interpret the language of regular expressions and to convert it into a form that is easy to apply quickly to a string. The reason to have a `Matcher` class (rather than just writing something like `P.find (s)`) is that you will often want to look for multiple matches for a pattern in `s`, and to see just what part of `s` was matched. Therefore, you need something that keeps track of both a regular expression, a string, and the last match within that string. The expression we used previously,

```
s.matches ("([b-df-hj-np-tv-z][aeiou]+\\s+)*")
```

where `s` is a `String`, is equivalent to

```
Pattern.compile ("([b-df-hj-np-tv-z][aeiou]+\\s+)*").matcher (s).matches ()
```

We can use complex regular expressions to do some limited *parsing*—breaking a string into parts. For example, suppose that we want to divide a time and date in the form “mm/dd/yy hh:mm:ss” (as in “3/7/02 12:36:12”) into its constituent parts. The means to do so is illustrated here:

```
Pattern daytime = Pattern.compile ("(\\d+)/ (\\d+)/ (\\d+)\\s+(\\d+):(\\d+):(\\d+)");
Matcher m = daytime.matcher ("It is now 3/7/02 12:36:12.");
if (m.find ()) {
    System.out.println ("Year: " + m.group(3));
    System.out.println ("Month: " + m.group(1));
    System.out.println ("Day: " + m.group(2));
    System.out.println ("Hour: " + m.group(4));
    System.out.println ("Minute: " + m.group(5));
    System.out.println ("Second: " + m.group(6));
}
```

That is, after any kind of matching operation (in this case, `find`), `m.group(n)` returns whatever was matched by the regular expression in the  $n^{th}$  pair of parentheses (counting from 1), or `null` if nothing matched that group.

Regular expressions and `Matchers` provide many more features than we have room for here. For example, you can control much more closely how many repetitions of a subpattern match a given string. You can not only find matches for patterns, but also conveniently replace those matches with substitute strings. The intent of this section was merely to suggest what you’ll find in the on-line documentation for the Java library.

## 9.6 Scanner

One application for text pattern matchers is in reading text input. A `Scanner`, found in package `java.util`, is a kind of all-purpose input reader, similar to a `Matcher`. In effect, you attach a `Scanner` to a source of characters—which can be a file, the standard input stream, or a string, among other things—and it acts as a kind of “nozzle” that spits out chunks of the stream as `Strings`. You get to select what constitutes a “chunk.” For example, suppose that the standard input consists of words alternating with integers, like this:

```
axolotl 12   aardvark 50
bison
16
cougar 6
panther 2 gazelle 100
```

and you want to read in these data and, let's say, print them out in a standard form and find the total number of animals. Here's one way you might do so, using a default `Scanner`:

```
import java.util.*;

class Wildlife {

    public static void main (String[] args) {
        int total;
        Scanner input = new Scanner (System.in);
        total = 0;
        while (input.hasNext ()) {
            String animal = input.next ();
            int count = input.nextInt ();
            total += count;
            System.out.format ("%15s %4d\n", animal, count);
        }
        System.out.format ("%15s %4d\n", "Total", total);
    }

}
```

This program first creates a `Scanner` out of the standard input (`System.in`). Each call to the `next` first skips any *delimiter* that happens to be there, and then returns the next *token* from the input: all characters up to and not including the next delimiter or the end of the input. Each call to `nextInt` is the same, but converts its token into an `int` first (and causes an error if the next token is not an `int`). Finally, `hasNext` returns true iff there is another token before the end of the input. For the given input, the program above prints

```
axolotl      12
aardvark     50
bison        16
cougar        6
panther       2
gazelle     100
Total       186
```

By default, the delimiter used is “any span of whitespace,” which is defined as one or more characters for which the method `Character.isWhitespace` returns true. You can change this. For example, to allow comments in the input file above, like this:

```
# Counts from sector #17
axolotl 12   aardvark 50
bison
```



```
16
cougar 6 # NOTE: count uncertain
panther 2 gazelle 100
```

insert the following line immediately after the definition of `input`:

```
input.useDelimiter (Pattern.compile ("(\\s|\\#.*)+"));
```

Just as with a `Matcher`, you can set a `Scanner` loose looking for a token that matches a particular pattern (rather than looking for whitespace that matches). So, for example, to skip to the next point in the input file that reads “Chapter *N*,” where *N* is an integer, you could write

```
if (input.findWithinHorizon ("Chapter \\d+", 0) != null) {
    /* What to do if you find a new chapter */
}
```

The second argument limits the number of characters searched; 0 means that there is no limit. This method returns the next match to the pattern string, or null if there is none.

Again, this section merely suggests what **Scanners** can do. The on-line documentation for the Java library contains the details.



## Chapter 10

# Generic Programming

Inheritance gives us one way to re-use the contents of a class, method, or interface by allowing it to mold itself to a variety of different kinds of objects having similar properties. However, it doesn't give us everything we might want.

For example, the Java library has a selection of useful classes that, like arrays, represent indexed sequences of values, but are expandable (unlike an array), and provide, in addition, a variety of useful methods. They are all various implementations of the interface `List`. One of these classes is `ArrayList`, which you might use as shown in the example here. Suppose we have a class called `Person`, designed to hold information such as name, Social-Security Number, and salary:

```
public class Person {
    public Person (String name, String SSN, double salary) {
        this.name = name; this.SSN = SSN; this.salary = salary;
    }

    public double salary;
    public String name, SSN;
}
```

We can create a list of `Persons` using a program along the following lines:

```
ArrayList people = new ArrayList ();
while (/*( there are more people to add )*/) {
    Person p = /*( the next person )*/;
    people.add (p);
}
```

That is, `.add` expands the list `people` by one object (which goes on the end). Then we could compute the total payroll like this:

```
double total;
total = 0.0;
for (int i = 0; i < people.size (); i += 1) {
    total += ((Person) people.get (i)).salary;
}
```

That is, `people.get(i)` is sort of like `A[i]` on arrays. But there's an odd awkwardness here: if `people` were an array, we'd write `people[i].salary`. What's with the `(Person)` in front? It is an example of a *cast*, a conversion, which we've seen in other contexts. For reference types, a cast  $(C) E$  means "I assert that the value of  $E$  is a  $C$ , and if it isn't, throw an exception."

To understand why we need this cast, we have to look at the definition of `ArrayList`, which (effectively) contains definitions like this<sup>1</sup>:

```
public class ArrayList implements List {
    /** The number of elements in THIS. */
    public int size () { ... }
    /** The Ith element of THIS. */
    public Object get (int i) { ... }
    /** Append X as the last element in THIS. (Always returns true). */
    public boolean add (Object x) { ... }
    ...
}
```

In order to make the class work on all kinds of objects, the type returned by `get` and taken by `add` is `Object`, the supertype of all reference types. Unfortunately, the Java compiler insists on knowing, for an expression such as `someone.salary`, that `someone` is actually going to have a `salary`. If all it knows is that `people.get(i)` is an `Object`, then it has no idea whether it is a kind of `Object` with a `salary`. Hence the need for the cast to `Person`. When you write

```
((Person) people.get (i)).salary
```

you are saying "I claim that the result of `people.get (i)` is a `Person`" and as a result, the Java system will check that the value returned by `get` really is a `Person` before attempting to take its `salary`. Furthermore, the compiler will proceed under the assumption that the value is a `Person`, and so must indeed have a `salary` field.

## 10.1 Simple Type Parameters

From your point of view, supplying this extra cast to `Person`, while not terribly burdensome, is redundant, adds to "code clutter," and also delays the detection of certain mistakes until the program executes. The only obvious way around it is unsatisfactory: creating new versions of `ArrayList` for every possible kind of element. Inheritance is no help, as it does not change the signatures (argument and return types) of methods. With Java 2, version 1.5, however, there is a way to say what you mean here—that `people` is a list of `Persons`—using *generic programming*:

---

<sup>1</sup>The return value of `add` indicates whether the operation changed the list, and is therefore useless in lists, where the `add` operation *always* changes the list. It's there because `List` is just one subtype of an even larger family of types in which the `add` method sometimes does *not* change the target object.

```

ArrayList<Person> people = new ArrayList<Person> ();
while (/*( there are more people to add )*/) {
    Person p = /*( the next person )*/;
    people.add (p);
}
...
double total;
total = 0.0;
for (int i = 0; i < people.size (); i += 1) {
    total += people.get (i).salary;
}

```

To make this work, the definition of `ArrayList` actually reads:

```

public class ArrayList<Item> implements List<Item> {
    /** The number of elements in THIS. */
    public int size () { ... }
    /** The Ith element of THIS. */
    public Item get (int i) { ... }
    /** Append X as the last element in THIS.  (Always returns true). */
    public boolean add (Item x) { ... }
    ...
}

```

The identifier `Item` here is a *type variable*, meaning that it ranges over reference types (interfaces, classes, and array types). The clause “`class ArrayList<Item>`” declares `Item` as a *formal type parameter* to the class `ArrayList`. Finally, `ArrayList<Person>` is a *parameterized type* in which `Person` is an *actual type parameter*. As you can see, you can use `ArrayList<Person>` just as you would an ordinary type—in declarations, **implements** or **extends** clauses, and with **new**, for example.

As another example, the Java library also defines the type `java.util.HashMap`, one of a variety of classes implementing `java.util.Map`, which is intended as a kind of dictionary type. Here is an example in which we take our array of `Persons` from above and arrange to be able to retrieve any `Person` record by name:

```

Map<String, Person> dataBase = new HashMap<String, Person> ();
for (int i = 0; i < people.size (); i += 1) {
    Person someone = people.get (i);
    dataBase.put (someone.name, someone);
}

```

Having set up `dataBase` in this fashion, we can now write little programs like this to retrieve employee records:

```

/** Read name queries from INPUT and respond by printing associated
 * information from DATA. Each query is a name (a string). The
 * end of INPUT or a single period (.) signals the end. */
public static void doQueries (Scanner input, Map<String, Person> data)
{
    while (input.hasNext ()) {
        String name = input.next ();
        if (name.equals ("."))
            break;
        Person info = data.get (name);
        System.out.format ("Name: %s, SSN: %s, salary: $%6.2f%n",
                           name, info.SSN, info.salary);
    }
}

```

The definition of `HashMap` needed to get this effect might look like this<sup>2</sup>:

```

public class HashMap<Key, Value> implements Map<Key, Value> {
    ...
    /** The value mapped to by K, or null if there is none. */
    public Value get (Object k) { ... }
    /** Cause get(K) to become V, returning the previous value of
     * get(K). Values for all other keys are unchanged. */
    public Value put (Key k, Value v) { ... }
}

```

The difference from the `ArrayList` example is that we have more than one type parameter.

## 10.2 Type Parameters on Methods

Suppose that you'd like to generalize the following method:

```

/** Set all the items in A to X. */
public void set (String[] A, String x)
{
    for (int i = 0; i < A.length; i += 1)
        A[i] = x;
}

```

This works for arrays of `Strings`, but nothing else. The rules of Java allow you to write

---

<sup>2</sup>You might wonder why the argument to `get` can be any `Object`. Since the argument to `put`, which is what allows you to add things to the map, is `Key`, `get(x)` will return `null` whenever `x` does not have type `Key`. This is harmless, however, and the designers, I guess, decided to be lenient.

```

/** Set all the items in A to X. */
public static void set2 (Object[] A, Object x)
{
    for (int i = 0; i < A.length; i += 1)
        A[i] = x;
}

```

because all arrays are subtypes of `Object[]`. However you won't find out until execution time if you have supplied an argument for `x` that doesn't have a matching type. With type parameters, however, you can write

```

/** Set all the items in A to X. */
public static <AType> void set3 (AType[] A, AType x)
{
    for (int i = 0; i < A.length; i += 1)
        A[i] = x;
}

```

Here `AType` is a formal type parameter of the *method*, which you can now call with any kind of array. The compiler will insure that you have supplied the right kind of value for `x`.

## 10.3 Restricting Type Parameters

The useful class `java.util.Collections` contains a number of methods that work on lists, among them one for sorting a list. With what you saw in the last section, you might think that this would suffice to sort any kind of list:

```

public static <T> void sort(List<T> list) {
    ...
}

```

Unfortunately, not quite. In order to sort anything, you have to know what it means for one item to be less than another, and there is no general method for determining such a thing. That is, somewhere in the body of `sort`, there will presumably have to be some statement like

```

if (list.get (i).lessThan (list.get (j))) etc.

```

but there is no such general `lessThan` method that works on all types.

By convention, Java classes can define an ordering upon themselves by implementing the `java.lang.Comparable` interface, which looks like this:

```

public interface Comparable<T> {
    /** Returns a value < 0 if THIS is 'less than' Y, > 0 if THIS is
     *  greater than Y, and 0 if equal. */
    int compareTo(T y);
}

```

So we really want to say that `sort` accepts a type parameter `T`, but only if `T` implements the `Comparable` interface. Here's one way to write it:

```
public static <T extends Comparable<T>> void sort(List<T> list) {
    ...
}
```

we call the “`extends Comparable<T>`” clause a *type bound*.

## 10.4 Wildcards

Sometimes, it really doesn't matter *what* kinds of items are stored in your `List`. For example, because `toString` is a method that works on all objects, if you want a method to print them all in a numbered list,

```
public static <T> void printAll (List<T> list) {
    for (int i = 1; i <= list.size (); i += 1)
        System.out.println (i + ". " + list.get (i));
}
```

(The ‘+’ operation on `Strings` allows you to concatenate any `Object` to a `String`; it does so by calling the `toString` on that `Object`.) However, you never really use the name `T` for anything. Therefore, you can use this shorthand instead:

```
public static void printAll (List<?> list) {
    for (int i = 1; i <= list.size (); i += 1)
        System.out.println (i + ". " + list.get (i));
}
```

You can restrict this *wildcard type variable*, just as you can named type variables:

```
public static void printAllNames (List<? extends Person> list) {
    for (int i = 1; i <= list.size (); i += 1)
        System.out.println (i + ". " + list.get (i).name);
}
```

## 10.5 Generic Programming and Primitive Types

One possibly unfortunate characteristic of Java is that primitive types (**boolean**, **long**, **int**, **short**, **byte**, **char**, **float**, and **double**) are completely distinct from reference types. Everything other than a primitive value is an `Object` (that is, all reference types are subtypes of `Object`).

Alas, all the examples of useful classes from the Java library that we've seen deal with `Objects`: you can have lists of `Objects` and maps from `Objects` to `Objects`, but you can't use the same classes to get lists of **ints**. The language is not set up to provide such a thing without essentially copying of these classes from the library



and making changes to them by hand—one for each primitive type. The designers thought this was not a great idea and came up with a compromise instead.

Package `java.lang` of the Java library contains a set of *wrapper classes* in `java.lang` corresponding to each of the primitive types:

<i>Primitive</i> $\Rightarrow$ <i>Wrapper</i>	<i>Primitive</i> $\Rightarrow$ <i>Wrapper</i>
<b>long</b> $\Rightarrow$ Long	<b>char</b> $\Rightarrow$ Character
<b>int</b> $\Rightarrow$ Integer	<b>boolean</b> $\Rightarrow$ Boolean
<b>short</b> $\Rightarrow$ Short	<b>float</b> $\Rightarrow$ Float
<b>byte</b> $\Rightarrow$ Byte	<b>double</b> $\Rightarrow$ Double

The wrapper classes turned out to be a convenient dumping ground for a variety of static methods and constants associated with the primitive types. For example, there are constants `MAX_VALUE` and `MIN_VALUE` defined in the wrappers for numeric types, so that `Integer.MAX_VALUE` is the largest possible `int` value. There are also methods for parsing `String` values containing numeric literals back and forth to primitive types.

The original reason for introducing these classes, however, was so that their instances could contain a value of the corresponding primitive type (commonly called *wrapping* or *boxing* the value.) That is,

```
Integer I = new Integer (42);
int i = I.intValue (); // i now has the value 42.
```

There is no way to change the `intValue` of a given `Integer` object once it is created; `Integers` are *immutable* just like primitive values. Since `Integer` is a reference type, however, we can store its instances in an `ArrayList`, for example:

```
/** A list of the prime numbers between L and U. */
public List<Integer> primes (int L, int U)
{
    ArrayList<Integer> primes = new ArrayList<Integer>();
    for (int x = L; x < U; x += 1)
        if (isPrime (x))
            primes.add (new Integer (x));
    return primes;
}
```

These wrappers let us use the library classes for primitive types, but the syntax is clumsy, especially if we want to perform primitive operations such as arithmetic:

```
/** Add X to each item in L. */
public void increment (List<Integer> L, int x)
{
    for (int i = 0; i < L.size (); i += 1)
        L.set (i, new Integer (L.get (i).intValue () + x));
}
```

So in version 1.5 of Java 2, the designers introduced *automatic boxing and unboxing* of primitive values. Basically, the idea is that a primitive value will be implicitly converted (*coerced*) to a corresponding wrapper value when needed, and vice-versa. For example:

```
Integer x = 42; // is equivalent to Integer x = new Integer (42)
int y = x;      // is equivalent to int y = x.intValue ();
```

Now we can write

```
/** Add X to each item in L. */
public void increment (List<Integer> L, int x)
{
    for (int i = 0; i < L.size (); i += 1)
        L.set (i, L.get (i) + x);
}
```

So now it at least *looks like* we can write classes that deal with all types of value, primitive and reference. Nothing, alas, is free. The boxing coercion especially uses up both space and time. For large and intensively used collections of values, it may still be necessary to tune one's program by building specialized versions of such general-purpose classes.

## 10.6 Formalities

Templates intrude in several places in the syntax. Let's take a look at the crucial definitions we postponed in previous chapters.

```
TypeArguments: < ActualTypeArgument+, >
ActualTypeArgument:
    ReferenceType
    Wildcard
Wildcard: ? WildcardBoundsopt
WildcardBounds:
    extends ReferenceType
    super ReferenceType
TypeParameters: < FormalTypeParameter+, >
FormalTypeParameter: SimpleName TypeBoundsopt
TypeBounds: extends Type&+
```

*TypeParameters* introduce new names for type parameters of classes, interfaces, and methods. They are like the formal parameters of a method. *TypeArguments*, on the other hand, are like the expressions that you supply in method call. For example, in this declaration from §10.1:

```
public class ArrayList<Item> implements List<Item> {
```

---

**Cross references:** *ReferenceType* 76, *Type* 76.

```
    ...
}
```

the clause `<Item>` after `ArrayList` is a single-element *TypeParameters*, and the `<Item>` after `List` is a *TypeArguments*.

A *FormalTypeParameter* defines a *SimpleName* to be a type that must be a subtype of all the *TypeBounds* supplied. The scope of this declaration is whatever class, interface, or method declaration includes it.

An *ActualTypeArgument* denotes some type that you are “plugging into” a *FormalTypeParameter*, rather as you would pass a parameter to a method. Unlike method calls, however, you are allowed to supply a *Wildcard*, or “don’t care” value for such a parameter. Given a declaration such as

```
class Widget<T> {
    private T x;
    void g (T y) { x = y; }
    T f () { return x; }
}
```

a declaration such as

```
Widget<?> w1;
```

means “`w1` is a `Widget`, which has an `f` method that returns some type, but I’m not going to tell you (the compiler) what that type is, so be ready for anything.” The compiler, being conservative, will assume nothing about the value `w1.f()`, except, of course, that it is some kind of `Object`.

If, on the other hand, I had written

```
Widget<? extends List<Integer>> w1;
```

I’d be telling the compiler that `w1.f()` returns some kind of `List<Integer>` (so that, for example, it knows to allow expressions like `w1.f().size()` or `w1.f().get(3) + 42`). Finally, something like

```
Widget<? super Integer> w1;
```

tells the compiler that `w1.g(3)` is legal, since even though it doesn’t know what static type `w1.x` has, it does know that it is a supertype of `Integer`.

**Some shorthand.** Declarations such as

```
ArrayList<Integer> primes = new ArrayList<Integer>();
```

are common enough that the repetition of `Integer` to the right of the assignment becomes tedious. In recent versions of Java, fortunately, you are allowed to shorten this in assignments and declarations where the type of the right-hand expression is clear:

```
ArrayList<Integer> primes = new ArrayList<>();
```

## 10.7 Caveats

Let's face it, inheritance and type parameterization in Java are both complicated. I've tried to finesse some of the complexity (for example, there are still a number of generic-programming features that I haven't discussed at all.) Here are a few particular pitfalls it might be helpful to beware of.

**Type hierarchy.** A very common mistake is to think that, for example, because `String` is a subtype of `Object`, that therefore `ArrayList<String>` must be a subtype of `ArrayList<Object>`. In fact, this is not the case. Actually, it's not difficult to see why. In Java, if  $C$  is a subtype of  $P$ , then anything you can do to a  $P$  you can also do to a  $C$ . If  $L$  is an `ArrayList<Object>`, then you can write

```
Person someone = ...;
L.add (someone);
```

But clearly, you *cannot* do that to an `ArrayList<String>`; therefore you'd better not be allowed to have  $L$  point to an `ArrayList<String>`, which means that `ArrayList<String>` had better *not* be a subtype of `ArrayList<Object>` (nor vice-versa).

**Consequences of the implementation.** You might be tempted to think that `ArrayList<Person>` is a just new copy of `ArrayList` with `Person` substituted for `Item` throughout. For a number of reasons, partly historical<sup>3</sup>, it's really just a plain `ArrayList`s under the hood and all varieties of `ArrayList<T>` for all  $T$  actually run exactly the same code. During translation, however, the compiler enforces the extra requirement that an `ArrayList<Person>` may only contain `Persons`. There are a couple of noticeable consequences of this design that are frankly annoying. If  $T$  is a type parameter, then

- You cannot use `new T`. For example, inside the definition of `ArrayList`, you can't write "`new Item[...]`."
- You cannot write `X instanceof T`.
- You cannot use  $T$  as the type of a static field.

[**Advanced Question:** See if you can figure out why these restrictions might exist.]

The restriction against using a type parameter in a `new` is bothersome. Suppose you'd like to write:

```
class MyThing<T> {
    private T[] stuff;
```

---

<sup>3</sup>These generic programming features were designed under stringent constraints. Backward compatibility, in particular, was an important goal. Old versions of `ArrayList` and other functions in the Java library had no parameters, and Java's designers did not want to have to force programmers to rewrite all their existing Java programs.

```
MyThing (int N) {  
    stuff = new T[N];  
}  
  
T get (int i) { return stuff[i]; }  
}
```

It makes perfect sense, but doesn't work, because you're not allowed to use `T` with `new`. Instead, write the following:

```
MyThing (int N) {  
    stuff = (T[]) new Object[N];  
}
```

This is a terrible kludge, really, justified only by necessity. Your author feels dirty just mentioning it. The code above would not work, for example, if you substituted a real type, like `String` for the type variable `T`; at execution time, you would get an error on the conversion to `String[]`, because an array of `Object` is not an array of `String`. The code above works only because the Java compiler “really” substitutes `Object` for `T` inside the body of `MyThing`, and then inserts additional conversions where needed. Perhaps it is advisable for the time being *not* to think about this unfortunate glitch too much and hope the designers eventually come up with a better solution!



## Chapter 11

# Multiple Threads of Control

Sometimes, we need programs that behave like several separate programs, running simultaneously and occasionally communicating. Familiar examples come from the world of graphical user interfaces (GUIs) where we have, for example, a chess program in which one part that “thinks” about its move while another part waits for its user to resize the board or push the “Resign” button. In Java, such miniprograms-within-programs are called *threads*. The term is short for “thread of control.” Java provides a means of creating threads and indicating what code they are to execute, and a means for threads to communicate data to each other without causing the sort of chaos that would result from multiple programs attempting to make modifications to the same variables simultaneously. The use of more than one thread in a program is often called *multi-threading*; however, we also use the terms *concurrent programming*, and *parallel programming*, especially when there are multiple processors each of which can be dedicated to running a thread.

Java programs may be executed on machines with multiple processors, in which case two active threads may actually execute instructions simultaneously. However, the language makes no guarantees about this. That’s a good thing, since it would be a rather hard effect to create on a machine with only one processor (as most have). The Java language specification also countenances implementations of Java that

- Allow one thread to execute until it wants to stop, and only then choose another to execute, or
- Allow one thread to execute for a certain period of time and then give another thread a turn (*time-slicing*), or
- Repeatedly execute one instruction from each thread in turn (*interleaving*).

This ambiguity suggests that threads are not intended simply as a way to get parallelism. In fact, the thread is a useful *program structuring tool* that allows sequences of related instructions to be written together as a single, coherent unit, even when they can’t actually be executed that way.

```
package java.lang;

public interface Runnable {
    /** Execute the body of a thread. */
    void run();
}

public class Thread implements Runnable {
    /** Minimum, maximum, and default thread priorities. */
    public static final int
        MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY;

    /** A new Thread with name NAME that will run
     *  the code in BODY.run() when started. */
    public Thread(Runnable body, String name);
    /** Same as Thread (BODY, some generated name) */
    public Thread(Runnable body);
    /** Same as Thread (null, NAME) */
    public Thread(String name);
    /** Same as Thread (this, some generated name) [that is,
     *  the Thread supplies its own run() routine.] */
    public Thread();

    /** Begin independent execution of the body of this
     *  Thread. Throws exception if this Thread has already
     *  started. */
    public void start();

    /** Inherited from Runnable. If this Thread provides
     *  its own body, this executes it. */
    public void run();

    /** The Thread that is executing the caller. */
    public static native Thread currentThread();
    /** Fetch (set) the name of this Thread. */
    public final String getName();
    public final void setName(String newName);
    /** The (set) the current priority of this Thread. */
    public final int getPriority();
    public final void setPriority(int newPriority);
```

*Continued on page 241.*

Figure 11.1: The interface `java.lang.Runnable` and class `java.lang.Thread`. See also §11.1.



*Class Thread continued from page 240.*

```

    /** Current interrupt state of this Thread. */
    public boolean isInterrupted();
    /** Causes this.isInterrupted() to become true. */
    public void interrupt();
    /** Resets this.isInterrupted() to false, and returns its
     *  prior value. */
    public static boolean interrupted();

    /** Causes CURRENT to wait until CURRENT.isInterrupted()
     *  or this thread terminates, or  $10^{-3} \cdot \text{MILLIS} + 10^{-9} \cdot \text{NANOS}$ 
     *  seconds have passed. */
    public final void join(long millis, int nanos)
        throws InterruptedException;
    /** Same as join (MILLIS, 0); join(0) waits forever. */
    public final void join(long millis)
        throws InterruptedException;
    /** Same as join(0) */
    public final void join() throws InterruptedException;

    /** Causes CURRENT to wait until CURRENT.isInterrupted()
     *  or  $10^{-3} \cdot \text{MILLIS} + 10^{-9} \cdot \text{NANOS}$  seconds have passed. */
    public static void sleep(long millis, int nanos)
        throws InterruptedException;
    /** Same as sleep(MILLIS, 0). */
    public static void sleep(long millis)
        throws InterruptedException;
}

```

Figure 11.2: Class `java.lang.Thread`, continued.

## 11.1 Creating and Starting Threads

Java provides the built-in type `Thread` (in package `java.lang`) to allow a program a way to create and control threads. Figures 11.1–11.2 describe this class and the `Runnable` interface as the programmer sees them. In the figure, `CURRENT` refers to the `Thread` executing the call, also called `Thread.currentThread()`.

There are essentially two ways to use it to create a new thread. Suppose that our main program discovers that it needs to have a certain task carried out—let’s say washing the dishes—while it continues with its work—let’s say playing chess. We can use either of the following two forms to bring this about:

```
class Dishwasher implements Runnable {
    Fields needed for dishwashing.
    public void run () { wash the dishes }
}

class MainProgram {
    public static void main (String[] args) {
        ...
        if (dishes are dirty) {
            Dishwasher washInfo = new Dishwasher ();
            Thread washer = new Thread (washInfo);
            washer.start ();
        }
        play chess
    }
}
```

Here, we first create a data object (`washInfo`) to contain all data needed for dishwashing. It implements the standard interface `java.lang.Runnable`, which requires defining a function `run` that describes a computation to be performed. We then create a `Thread` object `washer` to give us a handle by which to control a new thread of control, and tell it to use `washInfo` to tell it what this thread should execute. Calling `.start` on this `Thread` starts the thread that it represents, and causes it to call `washInfo.run ()`. The main program now continues normally to play chess, while thread `washer` does the dishes. When the `run` method called by `washer` returns, the thread is terminated and (in this case) simply vanishes quietly.

In this example, I created two variables with which to name the `Thread` and the data object from which the thread gets its marching orders. Actually, neither variable is necessary; I could have written instead

```
if (dishes are dirty)
    (new Thread (new Dishwasher ())).start ();
```

Java provides another way to express the same thing. Which you use depends on taste and circumstance. Here’s the alternative formulation:

```

class Dishwasher extends Thread {
    Fields needed for dishwashing.
    public void run () { wash the dishes }
}

class MainProgram {
    public static void main (String[] args) {
        ...
        if (dishes are dirty) {
            Thread washer = new Dishwasher ();
            washer.start ();
            // or just (new Dishwasher ()).start ();
        }
        play chess
    }
}

```

Here, the data needed for dishwashing are folded into the `Thread` object (actually, an extension of `Thread`) that represents the dishwashing thread.

When you execute a Java application (as opposed to an embedded Java program, such as an applet), the system creates an anonymous *main thread* that simply calls the appropriate `main` procedure. When the `main` procedure terminates, this main thread terminates and the system then typically waits for any other threads you have created to terminate as well.

## 11.2 A question of terminology

The fact that Java's `Thread` class has the name it does may tempt you into making the equation “thread = `Thread`.” Unfortunately, this is not correct. When I write “thread” (uncapitalized in normal text font), I mean “thread of control”—an independently executing instruction sequence. When I write “`Thread`” (capitalized in typewriter font), I mean “a Java object of type `java.lang.Thread`.” The relationship is that a `Thread` is an object visible to a program by which one can manipulate a thread—a sort of handle on a thread. Each `Thread` object is associated with a thread, which itself has no name, and which becomes active when some other thread executes the `start` method of the `Thread` object.

A `Thread` object is otherwise just an ordinary Java object. The thread associated with it has no special access to the associated `Thread` object. Consider a slight extension of the `Dishwasher` class above:

```

class SelfstartingDishwasher extends Dishwasher {
    SelfstartingDishwasher () {
        start ();
    }
}

class MainProgram {
    public static void main (String[] args) {
        ...
        if (dishes are dirty) {
            Thread washer = new SelfstartingDishwasher ();
            // actually, we could just use
            //     new SelfstartingDishwasher ();
            // and get an anonymous dishwasher, since we don't
            // refer to 'washer' again.
        }
        play chess
    }
}

```

Initially, the anonymous main thread executes the code in `MainProgram`. The main thread creates a `Thread` (actually a `SelfstartingDishwasher`), pointed to by `washer`, and then executes the statements in its constructor. That's right: the *main thread* executes the code in the `SelfstartingDishwasher` constructor, just as it would for any object. While executing this constructor, the main thread executes `start()`, which is short for `this.start()`, with the value `this` being that of `washer`. That activates the thread (lower case) associated with `washer`, and that thread executes the `run` method for `washer` (inherited, in this case, from `Dishwasher`). If we were to define additional procedures in `SelfstartingDishwasher`, they could be executed either by the main thread or by the thread associated with `washer`. The only magic about the type `Thread` is that among other things, it allows you to start up a new thread; otherwise `Threads` are ordinary objects.

Just to cement these distinctions in your mind, I suggest that you examine the following (rather perverse) program fragment, and make sure that you understand why its effect is to print 'true' a number of times. It makes use of the static (class) method `Thread.currentThread`, which returns the `Thread` associated with the thread that calls it. Since `Foo` extends `Thread`, this method is inherited by `Foo`.

```

// Why does the following program print nothing but 'true'?
class Foo extends Thread {
    public void run () {
        System.out.println (Main.mainThread != currentThread());
    }

    public void printStuff () {
        Thread mainThr = Main.mainThread;

```

```

        System.out.println (mainThr == Thread.currentThread ());
        System.out.println (mainThr == Foo.currentThread ());
        System.out.println (mainThr == this.currentThread ());
        System.out.println (mainThr == currentThread ());
    }
}

class Main {
    static Thread mainThread;
    public static void main (String[] args) {
        mainThread = Thread.currentThread ();
        Thread aFoo = new Foo ();
        aFoo.start ();
        System.out.println (mainThread == Foo.currentThread ());
        System.out.println (mainThread == aFoo.currentThread ());
        aFoo.printStuff ();
    }
}

```

### 11.3 Synchronization

Two threads can communicate simply by setting variables that they both have access to. This sounds simple, but it is fraught with peril. Consider the following class, intended as a place for threads to leave String-valued messages for each other.

```

class Mailbox { // Version I. (Non-working)
    volatile String msg;

    Mailbox () { msg = null; }

    void send (String msg0) {
        msg = msg0;
    }

    String receive () {
        String result = msg;
        msg = null;
        return result;
    }
}

```

Here, I am assuming that there are, in general, several threads using a `Mailbox` to send messages and several using it to receive, and that we don't really care which thread receives any given message as long as each message gets received by exactly one thread.

This first solution has a number of problems:

1. If two threads call `send` with no intervening call to `receive`, one of their messages will be lost (a *write-write conflict*).
2. If two threads call `receive` simultaneously, they can both get the same message (one example of a *read-write conflict* because we wanted one of the threads to *write* null into `result` before the other thread *read* it).
3. If there is no message, then `receive` will return `null` rather than a message.

Items (1) and (2) are both known as *race conditions*—so called because two threads are racing to read or write a shared variable, and the result depends (unpredictably) on which wins. We'd like instead for any thread that calls `send` to wait for any existing message to be received first before proceeding; likewise for any thread that calls `receive` to wait for a message to be present; and in all cases for multiple senders and receivers to wait their respective turns before proceeding.

### 11.3.1 Mutual exclusion

We could try the following re-writes of `send` and `receive`:

```
class Mailbox { Version II. (Still has problems)
    volatile String msg;

    Mailbox () { msg = null; }

    void send (String msg0) {
        while (msg != null)
            ; /* Do nothing */
        msg = msg0;
    }

    String receive () {
        while (msg == null)
            ; /* Do nothing */
        String result = msg;
        msg = null;
        return result;
    }
}
```

The `while` loops with empty bodies indulge in what is known as *busy waiting*. A thread will not get by the loop in `send` until `this.msg` becomes null (as a result of action by some other thread), indicating that no message is present. A thread will not get by the loop in `receive` until `this.msg` becomes non-null, indicating that a message is present. (For an explanation of the keyword `volatile`, see §11.3.3.)

Unfortunately, we still have a serious problem: it is still possible for two threads to call `send`, simultaneously find `this.msg` to be null, and then both set it, losing one message (similarly for `receive`). We want to make the section of code in each procedure from testing `this.msg` for null through the assignment to `this.msg` to be effectively *atomic*—that is, to be executed as an indivisible unit by each thread.

Java provides a construct that allows us to *lock* an object, and to set up regions of code in which threads *mutually exclude* each other if operating on the same object.

#### Syntax.

*SynchronizeStatement:*

**synchronized** ( *Expression* ) *Block*

The *Expression* must yield a non-null reference value.

**Semantics.** We can use this construct in the code above as follows:

```
class Mailbox { // Version III. (Starvation-prone)
    String msg;

    Mailbox () { msg = null; }

    void send (String msg0) {
        while (true)
            synchronized (this) {
                if (msg == null) {
                    msg = msg0;
                    return;
                }
            }
    }

    String receive () {
        while (true)
            synchronized (this) {
                if (msg != null) {
                    String result = msg;
                    msg = null;
                    return result;
                }
            }
    }
}
```

The two **synchronized** blocks above are said to *lock* the object referenced by their argument, `this`.

When a given thread, call it *t*, executes the statement

**synchronized** (*X*) { *S* }

the effect is as follows:

- If some thread other than *t* has one or more *locks* on the object pointed to by *X*, *t* stops executing until these locks are removed. *t* then places a lock on the object pointed to by *X* (the usual terminology is that it *acquires* a lock on *X*). It's OK if *t* already has such a lock; it then gets another. These operations, which together are known as *acquiring a lock on (the object pointed to by) X* are carried out in such a way that only one thread can lock an object at a time.
- *S* is executed.
- No matter how *S* exits—whether by **return** or **break** or exception or reaching its last statement or whatever—the lock that was placed on the object is then removed.

As a result, if every function that touches the object pointed to by *X* is careful to synchronize on it first, only one thread at a time will modify that object, and many of the problems alluded to earlier go away. We call *S* a *critical region* for the object referenced by *X*. In our Mailbox example, putting accesses to the **msg** field in critical regions guarantees that two threads can't both read the **msg** field of a Mailbox, find it null, and then both stick messages into it (one overwriting the other).

We're still not done, unfortunately. Java makes no guarantees about which of several threads that are waiting to acquire a lock will 'win'. In particular, one cannot assume a first-come-first-served policy. For example, if one thread is looping around in **receive** waiting for the **msg** field to become non-null, it can permanently prevent another thread that is executing **send** from ever acquiring the lock! The 'receiving' thread can start executing its critical region, causing the 'sending' thread to wait. The receiving thread can then release its lock, loop around, and re-acquire its lock even though the sending thread was waiting for it. We say that the sender can *starve*. There are various technical reasons for Java's being so loose and chaotic about who gets locks, which we won't go into here. We need a way for a thread to wait for something to happen without starving the very threads that could make it happen, and without monopolizing the processor with useless thumb-twiddling.

### 11.3.2 Wait and notify

The solution is to use the routines **wait**, **notify**, and **notifyAll** to temporarily remove locks until something interesting happens. These methods are defined on all **Objects**. If thread *T* calls *X.wait()*, then *T* must have a lock on the object pointed to by *X* (there is an exception thrown otherwise). The effect is that *T* temporarily removes all locks it has on that object and then *waits on X*. (Confusingly, this is *not* the same as waiting to acquire a lock on the object; that's why I used the phrase "stops executing" above instead.) Other threads can now acquire locks on



the object. If a thread that has a lock on *X* executes `notifyAll`, then all threads waiting on *X* at that time stop waiting and each tries to re-acquire all its locks on *X* and continue (as usual, only one succeeds; the rest continue to contend for a lock on the object). The `notify` method is the same, but wakes up only one thread (choosing arbitrarily). With these methods, we can re-code `Mailbox` as follows:

```
// Version IV. (Mostly working)
void send (String msg0) {
    synchronized (this) {
        while (msg != null)
            try {
                this.wait ();
            } catch (InterruptedException e) { }

        msg = msg0;
        this.notifyAll ();
    }
}

String receive () {
    synchronized (this) {
        while (this.msg == null)
            try {
                this.wait ();
            } catch (InterruptedException e) { }

        String result = msg;
        this.notifyAll ();    // Unorthodox placement.  See below.
        msg = null;
        return result;
    }
}
```

In the code above, the `send` method first locks the `Mailbox` and then checks to see if all messages that were previously sent have been received yet. If there is still an unreceived message, the thread releases its locks and waits. Some other thread, calling `receive`, will pick up the message, and notify the would-be senders, causing them to wake up. The senders cannot immediately acquire the lock since the receiving thread has not left `receive` yet. Hence, the placement of `notifyAll` in the `receive` method is not critical, and to make this point, I have put it in a rather non-intuitive place (normally, I'd put it just before the return as a stylistic matter). When the receiving thread leaves, the senders may again acquire their locks. Because several senders may be waiting, all but one of them will typically lose out; that's why the `wait` statements are in a loop.

This pattern—a function whose body synchronizes on `this`—is so common that there is a shorthand (whose meaning is nevertheless identical):

```

synchronized void send (String msg0) {
    while (msg != null)
        try {
            this.wait ();
        } catch (InterruptedException e) { }

    msg = msg0;
    this.notifyAll ();
}

synchronized String receive () {
    while (this.msg == null)
        try {
            this.wait ();
        } catch (InterruptedException e) { }

    String result = msg;
    this.notifyAll ();
    msg = null;
    return result;
}

```

There are still times when one wants to lock some arbitrary existing object (such as a shared **Vector** of information), and the **synchronized** statement I showed first is still useful.

It may have occurred to you that waking up all threads that are waiting only to have most of them go back to waiting is a bit inefficient. This is true, and in fact, so loose are Java's rules about the order in which threads get control, that it is still possible with our solution for one thread to be starved of any chance to get work done, due to a constant stream of fellow sending or receiving threads. At least in this case, *some* thread gets useful work done, so we have indeed improved matters. A more elaborate solution, involving more objects, will fix the remaining starvation possibility, but for our modest purposes, it is not likely to be a problem.

### 11.3.3 Volatile storage

You probably noticed the mysterious qualifier **volatile** in the busy-waiting example. Without this qualifier on a field, a Java implementation is entitled to pretend that no thread will change the contents of a field after another thread has read those contents until that second thread has acquired or released a lock, or executed a **wait**, **notify**, or **notifyAll**. For example, a Java implementation can treat the following two pieces of code as identical, and translate the one on the left into the one on the right:

```

while (X.z < 100) {
    tmp = X.z;
    while (tmp < 100) {

```

```

    X.z += 1;
}

    tmp += 1;
}
X.z = tmp;

```

But of course, the effects of these two pieces of code are *not* identical if another thread is also modifying `X.z` at the same time. If we declare `z` to be volatile, on the other hand, then Java must use the code on the left.

By default, fields are not considered volatile because it is potentially expensive (that is, slow) not to be able to keep things in temporary variables (which may often be fast registers). It's not common in most programs, with the exception of what I'll call the *shared, one-way flag idiom*. Suppose that you have a simple variable of some type other than `long` or `double`<sup>1</sup> and that some threads only read from this variable and others only write to it. A typical use: one thread is doing some lengthy computation that might turn out to be superfluous, depending on what other threads do. So we set up a boolean field in some object that is shared by all these threads:

```
volatile boolean abortComputation;
```

Every now and then (say at the top of some main loop), the computing thread checks the value of `abortComputation`. The other threads can tell it to stop by setting `abortComputation` to `true`. Making this field volatile in this case has exactly the same effect as putting every attempt to read or write the field in a `synchronized` statement, but is considerably cheaper.

## 11.4 Interrupts

Amongst programmers, the traditional definition of an *interrupt* is an “asynchronous transfer of control” to some pre-arranged location in response to some event<sup>2</sup>. That is, one's program is innocently tooling along when, between one instruction and the next, the system inserts what amounts to a kind of procedure call to a subprogram called an *interrupt handler*. In its most primitive form, an interrupt handler is not a separate thread of control; rather it usurps the thread that is running your program. The purpose of interrupts is to allow a program to handle a given situation when

---

<sup>1</sup>The reason for this restriction is technical. The Java specification requires that assigning to or reading from a variable of a type other than `long` and `double`, is *atomic*, which means that any value read from a variable is one of the values that was assigned to the variable. You might think that always has to be so. However, many machines have to treat variables of type `long` and `double` internally as if they were actually two smaller variables, so that assigning to them is actually *two* operations. If a thread reads from such a variable *during* an assignment (by another thread), it can therefore get a weird value that consists partly of what was previously in the variable, and partly what the assignment is storing into the variable. Of course, a Java implementation could “do the right thing” in such cases by quietly performing the equivalent of a `synchronized` statement whenever reading or writing such “big” variables, but that is expensive, and it was deemed better simply to let bad things happen and to warn programmers to be careful.

<sup>2</sup>Actually, ‘interrupt’ is also used to refer to the precipitating event itself; the terminology is a bit loose.

it occurs, without having to check constantly (or *poll*, to use the technical term) to see if the situation has occurred while in the midst of other work.

For example, from your program’s point of view, an interrupt occurs when you type a **Ctrl-C** on your keyboard. The operating system you are using handles numerous interrupts even as your program runs; one of its jobs is to make these invisible to your program.

Without further restrictions, interrupts are extremely difficult to use safely. The problem is that if one really can’t control at all where an interrupt might occur, one’s program can all too easily be caught with its proverbial pants down. Consider, for example, what happens with a situation like this:

```
// Regular program      | // Interrupt handler
theBox.send (myMessage); | theBox.send (specialMessage);
```

where the interrupt handlers gets called in the middle of the regular program’s **send** routine. Since the interrupt handler acts like a procedure that is called during the sending of **myMessage**, we now have all the problems with simultaneous sends that we discussed before. If you tried to fix this problem by declaring that the interrupt handler must wait to acquire a lock on **theBox** (i.e., changing the normal rule that a thread may hold any number of locks on an object), you’d have an even bigger problem. The handler would then have to wait for the regular program, but the regular program would not be running—the handler would be!

Because of these potential problems, Java’s “interrupts” are greatly constrained; in fact, they are really not asynchronous at all. One cannot, in fact, interrupt a thread’s execution at an arbitrary point, nor arrange to execute an arbitrary handler. If **T** points to an active **Thread** (one that has been started, and has not yet terminated), then the call **T.interrupt()** puts the thread associated with **T** in *interrupted status*. When that thread subsequently performs a **wait**—or if it is waiting at the time it is interrupted—two things happen in sequence:

- The thread re-acquires all its locks on the object on which it executed the **wait** call in the usual way (that is, it waits for any other thread to release its locks on the object).
- The thread then throws the exception **InterruptedException** (from package **java.lang**). This is a checked exception, so your program must have the necessary **catch** clause or **throws** clause to account for this exception. At this point, the thread is no longer in interrupted status (so if it chooses to wait again, it won’t be immediately re-awakened).

In Figures 11.1–11.2, methods that are declared to throw **InterruptedException** do so when the current thread (the one executing the call, not necessarily the **Thread** whose method is being executed) becomes interrupted (or was at the time of the call). When that happens **CURRENT.isInterrupted()** reset to false.

You’ve probably noticed that being interrupted requires the coöperation of the interrupted thread. This discipline allows the programmers to insure that interrupts occur only where they are non- disruptive, but it makes it impossible to achieve

effects such as stopping a renegade thread “against its will.” In the current state of the Java library, it appears that this is, in fact, impossible. That is, there is no general way, short of executing `System.exit` and shutting down the entire program, to reliably stop a runaway thread. While poking around in the Java library, you may run across the methods `stop`, `suspend`, `resume`, and `destroy` in class `Thread`. All of these methods are either *deprecated*, meaning that they are present for historical reasons and the language designers think you shouldn’t use them, or (in the case of `destroy`) unimplemented. I suggest that you join in the spirit of deprecation and avoid these methods entirely. First, it is very tricky to use them safely. For example, if you manage to `suspend` a thread while it has a lock on some object, that lock is not released. Second, because of the peculiarities of Java’s implementation, they won’t actually do what you want. Specifically, the `stop` method does not stop a thread immediately, but only at certain (unspecified) points in the program. Threads that are in an infinite loop doing nothing but computation may never reach such a point.



# Index

- !, *see* logical not
- !=, *see* not equals, *see* not equals
- &&, *see* conditional and, *see* conditional and
- &, *see* bitwise and, logical and
- \*, *see* multiply, *see* multiply
- \* (in grammars) , *see* Kleene star
- +, *see* unary plus, addition, concatenation, *see* unary plus, addition, concatenation, *see* unary plus, addition, concatenation
- + on `String`, *see* string concatenation
- + (in grammars), 64
- ++, *see* pre/post increment
- += on `String`, *see* string concatenation
- , *see* unary minus, subtraction, *see* unary minus, subtraction
- , *see* pre/post decrement
- ., *see* selection, *see* selection
- ..., *see* varargs
- /, *see* divide, *see* divide
- /\*(...)\*/ , 20
- /\*{...}\*/ , 20
- <, *see* comparison, *see* comparison
- <<, *see* shift left
- <=, *see* comparison, *see* comparison
- <> shorthand, 231
- ==, *see* equals, *see* equals
- >, *see* comparison, *see* comparison
- >=, *see* comparison, *see* comparison
- >>, *see* shift right arithmetic
- >>>, *see* shift right logical
- ?:, *see* conditional expression, *see* conditional expression
- @, *see* annotations
- [], *see* array indexing
- %, *see* remainder, *see* remainder
- main, 110
- ~, *see* complement
- ^, *see* exclusive or
- ||, *see* conditional or, *see* conditional or
- abrupt completion, 176
- abstract keyword, 52, 83, 96
- abstract class, 85
- abstract method, 98
- access control, 109
- accumulator, 35
- acquiring a lock, 244
- actual parameters, 6, 152
- actual type parameter, 225
- Actuals, 152
- ActualTypeArgument, 230
- addition, 125, 134
- Adjective, 63
- Adverb, 63
- algebraic language, 117
- Algol 60, 63
- Allocator, 139
- Annotation, 112
- AnnotationArguments, 112
- annotations, 112
- annotations (@), 112
- anonymous class, 93
- AnonymousClassAllocator, 93
- append (`StringBuilder`), 209
- applet, 6, 110
- array, 17--18, 38--39, 42--44, 68, 144
- multi-dimensional, 42--44
- array indexing, 144
- array indexing ([]), 118
- ArrayAccess, 144

- ArrayAllocator, 146
- ArrayIndexOutOfBoundsException class, 144, 184
- ArrayInitializer, 146
- ArrayType, 72
- ASCII, 194, 196
- Assignment, 116, 150
- assignment, 150
- AssignmentOperator, 116, 150
- association, 10, 117
- back trace, 58
- Backus-Naur Form, *see* BNF
- Backus-Normal Form, *see* BNF
- BasicClassDeclaration, 64, 83
- BasicInterfaceDeclaration, 87
- BinaryExpression, 116, 125, 128, 131, 138, 143, 147
- BinaryOperator, 116, 125, 128, 131, 138
- bitwise and, 128
- bitwise and, logical and (&), 118
- bitwise operators, 128
- bitwise or, 128
- bitwise or, logical or (|), 118
- blank final variables, 161
- Block, 164
- block structure, 76
- BlockStatement, 164
- BNF, 63
- boolean keyword, 72
- boolean values, 137
- BooleanLiteral, 138
- boxing, 230
- break keyword, 17, 33, 171, 176
- BreakStatement, 176
- BufferedReader class, 185
- byte keyword, 72
- Call, 152
- call back, 95
- call convertible, 99
- CASE\_INSENSITIVE\_ORDER, 199, 200
- case keyword, 165
- Cast, 119
- cast, 119
- casts, 118
- catch keyword, 181
- char keyword, 72
- Character class, 197
- character literals, 123
- character set, 194
- CharacterLiteral, 124
- charAt (String), 13, 199, 203
- charAt (StringBuilder), 208
- checked exception, 188
- class keyword, 44, 83, 157
- Class class, 157
- class literal, 157
- class method, 13, 84, 98
- Class methods
  - forName, 158
  - newInstance, 158
- class path, 79
- class variable, 84
- class variables, 86
- class with no instances (idiom), 107
- class, anonymous, 93
- class, enum, 88
- class, inner, 90
- class, local, 91
- class, member, 90
- class, nested, 90
- ClassAllocator, 140
- ClassBody, 83
- ClassBodyDeclaration, 83
- ClassDeclaration, 83
- ClassModifier, 83
- ClassOrInterfaceType, 72
- ClassType, 72
- clone (Object), 108
- closure, 94
- coercion, 119
- comments, 6, 65
- compareTo (String), 200, 205
- compareToIgnoreCase (String), 200
- comparison (<=), 11, 118
- comparison (<), 11, 118
- comparison (>=), 11, 118
- comparison (>), 11, 118
- comparison operators, 138



- CompilationUnit, 110
- compile-time constant, 156
- compiler, 7
- complement, 128
- complement (~), 118
- concat (String), 201, 202
- concurrent programming, 235
- conditional and, 138
- conditional and (&&), 11, 118
- conditional expression, 139
- conditional expression (?,:), 15, 118
- conditional or, 138
- conditional or (||), 11, 118
- conditional statement, 165
- constant expression, 156
- ConstantDeclaration, 87
- ConstantExpression, 156
- ConstantModifier, 87
- constructor, 47--48, 103
- ConstructorDeclaration, 103
- ConstructorModifier, 103
- container, 68
  - anonymous, 68
  - definition, 67
  - labeled, 68
  - simple, 68
  - structured, 68, 144
- context-free syntax, 62
- continue keyword, 176, 178
- ContinueStatement, 178
- control characters, 194
- conversion, 119
- conversion, explicit, 119
- conversion, implicit, 120
- conversion, widening, 120
- copy constructor, 107
- core of a language, 61
- critical region, 244
- dangling else, 166
- DecimalDigit, 122
- declaration, 75
- declarative region, 76
- Declarator, 81
- default keyword, 165
- default access, 56, 109
- default constructor, 104
- definite assignment, 159
- delete (StringBuilder), 210
- deleteCharAt (StringBuilder), 210
- dereference, 70
- design patterns
  - Singleton, 107
- destroy (Thread), 249
- Dim, 72, 146
- DimExpr, 146
- direct subclass, 85
- direct superclass, 85
- direct superinterface, 87
- divide, 125, 134
- divide (/), 9, 118
- do keyword, 32, 170
- documentation comment, 14, 65
- double keyword, 72
- Double methods
  - parseDouble, 18
- Double.MAX\_VALUE field, 133
- Double.MAX\_VALUE fields
  - MAX\_VALUE, 133
- Double.MIN\_VALUE field, 133
- Double.MIN\_VALUE fields
  - MIN\_VALUE, 133
- Double.NaN field, 133
- Double.NaN fields
  - NaN, 133
- Double.NEGATIVE\_INFINITY field, 133
- Double.NEGATIVE\_INFINITY fields
  - NEGATIVE\_INFINITY, 133
- Double.POSITIVE\_INFINITY field, 133
- Double.POSITIVE\_INFINITY fields
  - POSITIVE\_INFINITY, 133
- dynamic
  - property, 62
  - semantics, 62
- dynamic method set, 100
- dynamic type, 72
- ElementInitializer, 146
- ElementValue, 112
- ElementValuePair, 112

- else keyword, 165
- EmptyStatement, 164
- endsWith (String), 200
- enum, 88
- EnumBodyDeclarations, 88
- EnumConstant, 88
- EnumDeclaration, 88
- enumerals, 88
- enumerals, 88
- enumerated class, 88
- environment, 73
- equality operators, 138
- equals (String), 13, 200, 205
- equals (==), 11, 118
- equalsIgnoreCase (String), 200
- Eratosthenes, sieve of, 40--42
- Error class, 188
- EscapeSequence, 124
- evaluation order, 117
- exception, 58--59
  - checked, 188
  - unhandled, 184
- exclusive or, 128, 138
- exclusive or (^), 118
- exiting methods, 179
- exiting statements, 176
- explicit conversion, 119
- ExplicitConstructorCall, 103
- Exponent, 133
- Expression, 115
- expression
  - definition, 115
  - language, 115
- ExpressionList, 64
- ExpressionName, 78
- Extends, 83
- extends keyword, 52, 83, 87
  
- false keyword, 138
- field access, 140
- FieldAccess, 140
- FieldDeclaration, 86, 105, 108
- FieldModifier, 86
- fields, 86
- FileNotFoundException class, 185
- FileReader class, 185
- fillInStackTrace (Throwable), 190
- final keyword, 17, 81--83, 85--87, 96, 98, 161
- final methods, 98
- final variable, 82
- Finally, 181
- finally keyword, 181, 186
- first-class values, 157
- float keyword, 72
- Float.MAX\_VALUE field, 133
- Float.MAX\_VALUE fields
  - MAX\_VALUE, 133
- Float.MIN\_VALUE field, 133
- Float.MIN\_VALUE fields
  - MIN\_VALUE, 133
- Float.NaN field, 133
- Float.NaN fields
  - NaN, 133
- Float.NEGATIVE\_INFINITY field, 133
- Float.NEGATIVE\_INFINITY fields
  - NEGATIVE\_INFINITY, 133
- Float.POSITIVE\_INFINITY field, 133
- Float.POSITIVE\_INFINITY fields
  - POSITIVE\_INFINITY, 133
- floating-point literal, 9
- FloatType, 133
- for keyword, 35, 170
- ForCondition, 170
- ForInit, 170
- formal parameter, 14, 97
- formal type parameter, 225
- Formals, 96
- FormalTypeParameter, 230
- format (String), 199, 202
- format effectors, 194
- format specifier, 202
- format string, 202
- forName (Class), 158
- ForUpdate, 170
- free format, 65
- function closure, 94
  
- gcd (greatest common divisor), 31
- getChars (StringBuilder), 208

- getClass (Object), 157
- getLocalizedMessage (Throwable), 190
- getMessage (Throwable), 190
- grammar, 62
- greatest common divisor, 31
- grouping, 117
- grouping of operands, 10
- guarded commands, 20
- Handler, 181
- hashCode (String), 201, 206
- HexDigit, 122
- hiding, 76
- hiding of methods, 100
- hole in scope, 76
- Identifier, 75
- identifier, 75
- idiom
  - class with no instances, 107
  - copy constructor, 107
  - singleton object, 107
- if keyword, 15, 165
- iff, 20
- IllegalArgumentException class, 182
- Implements, 83
- implements keyword, 51, 83, 87
- import
  - package, 81
  - static, 81
  - type, 81
- import keyword, 57, 80
- ImportClause, 80
- IncrDecr, 116, 150
- indexing, array, 144
- indexOf (String), 25, 201
- indirect calls, 152
- infinity, 10, 132
- infix notation, 117
- inheritance, 52--56, 85
- InitDeclarator, 81, 105, 108
- inner class, 90
- inner class, allocating, 91
- insert (StringBuilder), 209
- instance method, 46--47, 98
- instance method call, 154
- instance variables, 86
- Initializer, 105
- instanceof keyword, 118, 143
- int keyword, 72
- integer literal, 9
- integer literals, 122
- Integer methods
  - parseInt, 18
- IntegerLiteral, 122
- IntegerNumeral, 122
- IntegerTypeSuffix, 122
- interface, 49--52, 87
- interface keyword, 87
- interface, member, 90
- interface, nested, 90
- InterfaceBodyDeclaration, 87
- InterfaceDeclaration, 87
- InterfaceModifier, 87
- InterfaceType, 72
- intern (String), 201, 205
- interpolation, 39
- interpreter, 8
- InterruptedException class, 248
- interrupts, 247
- IOException class, 185
- Iterable class, 175
- iteration, 31
- IterationStatement, 170
- Iterator class, 173
- iterators, 173
- java (interpreter), 8
- java.io classes
  - BufferedReader, 185
  - FileNotFoundException, 185
  - FileReader, 185
  - IOException, 185
  - PrintStream, 214--215
  - PrintWriter, 214--215
- java.lang package, 79
- java.lang classes
  - ArrayIndexOutOfBoundsException, 144, 184
  - Character, 197
  - Class, 157

- Error, 188
- IllegalArgumentException, 182
- InterruptedException, 248
- NegativeArraySizeException, 146
- NullPointerException, 140, 141, 144, 155, 182
- NumberFormatException, 184
- Object, 85
- Runnable, 236
- RuntimeException, 188
- String, 147--149, 197--206
- StringBuilder, 206--207
- Thread, 236, 238
- Throwable, 182, 190
- java.lang.StringBuilder constructor, 208
- java.lang.StringBuilder methods
  - StringBuilder (constructor), 208
- java.util classes
  - Iterable, 175
  - Iterator, 173
  - Scanner, 219--221
- java.util.regex classes
  - Matcher, 216
  - Pattern, 215--219
- javac (compiler), 7
- JumpStatement, 176
- keywords, 75
- Kleene star, 64
- Label, 176
- LabeledStatement, 176
- lastIndexOf (String), 201
- left associative operator, 117
- left-to-right evaluation, 117
- LeftHandSide, 116, 150
- length field, 144
- length (String), 13, 199, 203
- length (StringBuilder), 208
- Letter, 75
- LetterOrDigit, 75
- lexical structure, 61
- library, standard, 62
- linear interpolation, 39
- lists (in grammars), 64
- Literal, 118
- local class, 91
- local variable, 81
- LocalClassDeclaration, 91
- LocalDeclarationStatement, 81
- lock, 243
- lock, acquiring, 244
- logical and, 138
- logical not, 138
- logical not (!), 118
- logical or, 138
- long keyword, 72
- loop
  - and-a-half, 171
  - do...while, 32, 170
  - for, 170
  - while, 32, 170
- loops, restarting, 178
- main program, 6, 18, 110
- Matcher class, 216
- Math methods
  - sqrt, 14
- MAX\_VALUE (field), 133
- MemberName, 78
- members of a class, 6, 84
- MemberTypeDeclaration, 90
- MemberTypeModifier, 90
- method call, 6, 152
- method call, instance, 154
- method call, static, 153
- method declarations, 95
- method resolution, static, 153
- method signature, 14
- method, synchronized, 245
- MethodBody, 96
- MethodDeclaration, 95
- MethodModifier, 96
- MethodName, 78, 152
- methods, exiting, 179
- MethodSelector, 152
- MethodSignature, 95
- MethodSignatureDeclaration, 87
- MIN\_VALUE (field), 133
- modular arithmetic, 122, 125

- multi-dimensional array, 42--44
- multiply, 125, 134
- multiply (\*), 9, 118
- Name, 78
- NaN (field), 133
- NaN (Not A Number), 132
- NaN (Not a Number), 10
- native keyword, 96, 98
- negative 0, 132
- NEGATIVE\_INFINITY (field), 133
- NegativeArraySizeException class, 146
- nested class, 90
- nested interface, 90
- new keyword, 38, 93, 140, 146
- new T<> shorthand, 231
- newInstance (Class), 158
- newline, 124
- non-static member, 84
- nonterminal symbol, 63
- not equals (!=), 11, 118
- NounPhrase, 63, 64
- null keyword, 44, 140
- NullLiteral, 139
- NullPointerException class, 140, 141, 144, 155, 182
- NumberFormatException class, 184
- Object class, 85
- Object methods
  - clone, 108
  - getClass, 157
  - toString, 202
- object-based programming, 46
- OctalDigit, 122
- OctalEscape, 124
- Operand, 115, 139, 157
- OtherPrimary, 116
- outer-level types and packages, 79
- OuterObject, 93, 140
- overloading, 22, 76, 99
- overriding, 85, 100
- package, 56--57, 78
- package keyword, 79
- package representation, 79
- package-private access, 57, 109
- PackageClause, 79
- PackageImport, 80
- PackageName, 78
- parallel programming, 235
- Parameter, 96
- parameter, actual, 6, 152
- parameterized type, 225
- ParameterList, 64
- parenthesized expression, 117
- parseDouble (Double), 18
- parseInt (Integer), 18
- pass by value, 153
- Pattern class, 215--219
- pig latin, 24
- pointer, 38, 68
  - null, 70
- POSITIVE\_INFINITY (field), 133
- PositiveDigit, 122
- PositiveOctalDigit, 122
- postfix operator, 117
- PostfixExpression, 116, 150
- pre/post decrement (--), 118
- pre/post increment (++), 118
- precedence of operators, 10, 117
- prefix operator, 117
- Primary, 116
- prime number, 18
- prime sieve, 40--42
- PrimitiveType, 72
- print (PrintStream), 7
- println (PrintStream), 7
- printStackTrace (Throwable), 190
- PrintStream class, 214--215
- PrintStream methods
  - print, 7
  - println, 7
- PrintWriter class, 214--215
- private keyword, 13, 56, 86, 90, 96, 103, 109
- private constructors, 107
- programming language
  - definition, 61
- prologue, standard, 62
- proper subclass, 85

- proper subtype, 85
- proper superclass, 85
- protected keyword, 56, 86, 90, 96, 103, 109
- public keyword, 13, 56, 83, 86, 87, 90, 96, 103, 109
- public type, 85
- qualified name, 78
- Qualifier, 78
- RealLiteral, 132
- reduction loop, 35
- reference, 38, *see* pointer
- reference type, 38
- ReferenceType, 72
- referent, 70
- reflection, 156
- regionMatches (String), 200
- regular expression, 216
- remainder, 125, 134
- remainder (%), 9, 118
- replace (String), 201
- replace (StringBuilder), 210
- reserved words, 75
- resume (Thread), 249
- return keyword, 179
- ReturnStatement, 179
- reverse (StringBuilder), 210
- Runnable class, 236
- RuntimeException class, 188
- Scanner class, 219--221
- scope, 76
- selection, 6, 78
- selection (.), 6, 118
- selection statement, 165
- SelectionStatement, 165
- semantics, 62
- Sentence, 63
- setCharAt (StringBuilder), 210
- setLength (StringBuilder), 210
- shadowing, *see* hiding
- shift left, 128
- shift left (<<), 118
- shift right arithmetic, 128
- shift right arithmetic (>>), 118
- shift right logical, 128
- shift right logical (>>>), 118
- short keyword, 72
- side effect, 115
- sieve of Eratosthenes, 40--42
- Sign, 133
- signature, method, 14
- SimpleName, 78
- SimpleNounPhrase, 64
- SingleCharacter, 124
- singleton object (idiom), 107
- SingularNoun, 63
- SingularVerb, 63
- sqrt (Math), 14
- stack trace, 58
- standalone application, 110
- standard library, 79
- standard prologue, 62
- start symbol, 63
- startsWith (String), 200
- starvation, 244
- Statement, 163
- statement
  - definition, 115
- StatementExpression, 116
- StatementExpressions, 170
- static
  - property, 61
  - semantics, 62
- static keyword, 80, 86, 87, 90, 96, 108
- static field, 86
- static member, 84
- static method, 13, 98
- static method call, 153
- static method resolution, 153
- static nested type, 90
- static type, 72
- StaticImport, 80
- StaticInitializer, 108
- StaticMemberName, 78
- stop (Thread), 249
- Strings, immutability of, 197
- String class, 147--149, 197--206

- string concatenation, 12, 149
- String methods
  - charAt, 13, 199, 203
  - compareTo, 200, 205
  - compareToIgnoreCase, 200
  - concat, 201, 202
  - endsWith, 200
  - equals, 13, 200, 205
  - equalsIgnoreCase, 200
  - format, 199, 202
  - hashCode, 201, 206
  - indexOf, 25, 201
  - intern, 201, 205
  - lastIndexOf, 201
  - length, 13, 199, 203
  - regionMatches, 200
  - replace, 201
  - startsWith, 200
  - substring, 13, 199, 203
  - toLowerCase, 200
  - toString, 201
  - toUpperCase, 200
  - trim, 201
  - valueOf, 199
- StringBuilder (constructor), 208
- StringBuilder class, 206--207
- StringBuilder methods
  - append, 209
  - charAt, 208
  - delete, 210
  - deleteCharAt, 210
  - getChars, 208
  - insert, 209
  - length, 208
  - replace, 210
  - reverse, 210
  - setCharAt, 210
  - setLength, 210
  - substring, 208
  - toString, 208
- StringCharacter, 147
- StringLiteral, 147
- strings, 147, 197
- structured container, 144
- subclass, 85
- substring (String), 13, 199, 203
- substring (StringBuilder), 208
- subtraction, 125, 134
- subtype, 51, 85, 87, 88
- super keyword, 103, 140, 141, 152
- superclass, 85
- superinterface, 87
- SuperInterfaces, 87
- suspend (Thread), 249
- switch keyword, 16, 165
- SwitchBlockStatements, 165
- SwitchLabel, 165
- synchronized keyword, 96, 243, 245
- synchronized method, 245
- SynchronizeStatement, 243
- syntactic variable, 63
- syntax, 62
- tail recursion, 30
- terminal symbols, 63
- This, 154
- this keyword, 46, 91, 98, 103, 154
- thread, 235
- Thread class, 236, 238
- Thread methods
  - destroy, 249
  - resume, 249
  - stop, 249
  - suspend, 249
- throw keyword, 181
- Throwable class, 182, 190
- Throwable methods
  - fillInStackTrace, 190
  - getLocalizedMessage, 190
  - getMessage, 190
  - printStackTrace, 190
  - toString, 190
- Throws, 96, 188
- throws keyword, 96, 188
- ThrowStatement, 181
- toLowerCase (String), 200
- toString (Object), 202
- toString (String), 201
- toString (StringBuilder), 208
- toString (Throwable), 190



- toUpperCase (String), 200
- transient keyword, 86
- trim (String), 201
- true keyword, 138
- try keyword, 181, 183
- TryStatement, 181
- Type, 72
- type, 70
  - definition, 67
  - dynamic, 72
  - static, 72
- type bound, 228
- type denotation, 72
- type parameter, 225
- type variable, 225
- type, testing, 143
- TypeArguments, 230
- TypeBounds, 230
- TypeDeclaration, 83
- TypeImport, 80
- TypeName, 78
- TypeParameters, 230
  
- unary minus, 125, 134
- unary minus, subtraction (-), 9, 118
- unary plus, 125, 134
- unary plus, addition, concatenation  
    (+), 9, 12, 118
- UnaryAdditiveOperator, 125, 131
- UnaryExpression, 116, 125, 128, 131,  
    138, 150
- UnaryOperator, 116
- unboxing, 230
- Unicode, 195
  
- value
  - definition, 67
- value parameters, 153
- valueOf (String), 199
- VarArgs, 96
- varargs, 96
- varargs (...), 96
- variable-arity parameter, *see* varargs
- variant, 22
- VarInit, 81
- VerbPhrase, 63
  
- virtual machine, 110
- void keyword, 13, 96
- volatile keyword, 86, 246
  
- while keyword, 32, 170
- whitespace, 65
- widening, 120
- Wildcard, 230
- WildcardBounds, 230
- wildcards, 228
- wrapper classes, 229
  
- ZeroToThree, 124