Assorted Materials on Java

Paul N. Hilfinger
University of California, Berkeley

Acknowledgments. Thanks to Soheil Golshan for pointing out errors in an earlier edition of these notes.

Contents

Chapter 1

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).

1.1 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[1].

- *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.

[1]This sentence is a famous example due to Noam Chomsky.

1.2 Describing Lexical Structure and Syntax

The original description of the Algol 60 language[2] 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)[3]. 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[4]:

Sentence: NounPhrase VerbPhrase
NounPhrase: Adjective NounPhrase
NounPhrase: Adjective PluralNoun
VerbPhrase: PluralVerb $Adverb_{opt}$
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 *Noun-Phrase* 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,

[2]P. Nauer, ed., "Report on the Algorithmic Language Algol 60," *Comm ACM*, 6(1), 1963, pp. 1–17.

[3]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.

[4]This example is adapted from Hopcroft and Ullman, *Formal Languages and Their Relation to Automata,* Addison-Wesley, 1969.

we don't mention *lexical* details such as blanks when giving grammar rules; we'll 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:* $\textit{Adjective}^{+}$ *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:* $\textit{Adjective}^{*}$ *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:* $\textit{ExpressionList}_{opt}$

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

> $\textit{Expression}^{+}_{,}$
> $\textit{Expression}^{*}_{,}$

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_opt Implements_opt
        { ClassBodyDeclaration* }
```

Chapter 2

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$^{\text{tm}}$ 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.

2.1 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 §2.2) 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.

2.1.1 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[1], each with a *label* or *name,* as in Figure 2.1a. Figures 2.1b–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 2.2 shows various alternative depictions of a sample array whose elements are labeled by integers and whose elements contain numbers.

2.1.2 Pointers

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

[1]The case of a structured container with no containers inside it is a bit unusual, I admit, but it does occur.

[2]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.

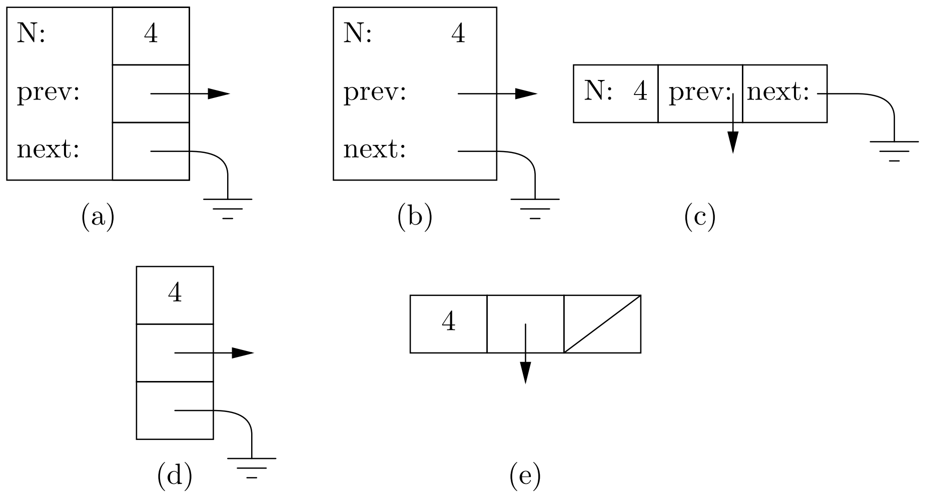


Figure 2.1: A structured container, depicted in several different ways. Diagrams (d) and (e) assume that the labels are known from context.

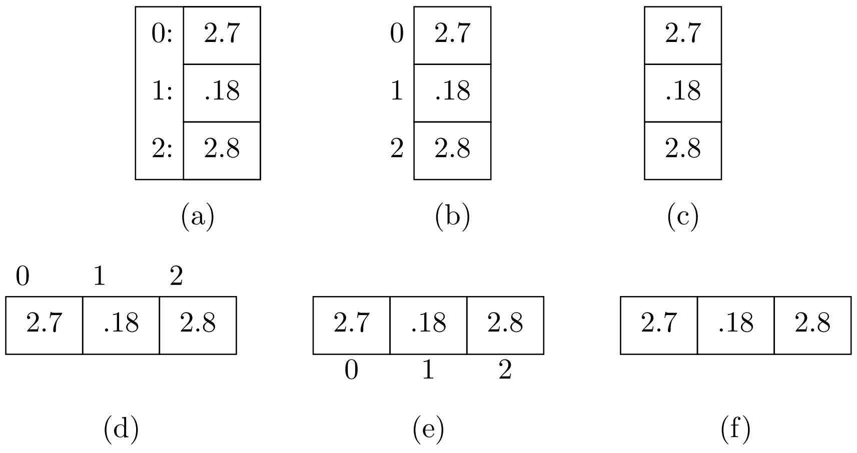
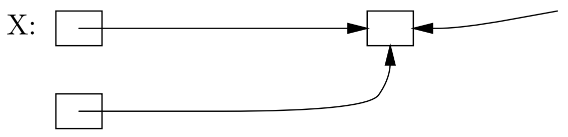
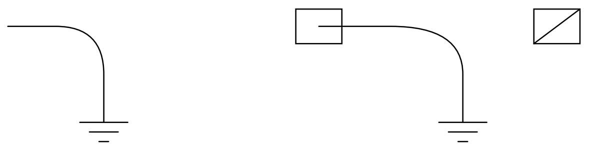


Figure 2.2: Various depictions of one-dimensional array objects. The full diagram, (a), is included for completeness; it is generally not used for arrays. The diagrams without indices, (c) and (f), assume that the indices are known from context or are unimportant.



(a) All pointers here are equal.



(b) Null pointers.

Figure 2.3: 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 2.3a 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 2.3b 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 2.2, 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 2.2a has the value `<2.7, 0.18, 2.8>`, and that assigning, say, 0 to the first element of the array replaces its *entire* contents with the value `<0, 0.18, 2.8>`. 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."

2.2 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!).

2.2.1 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.

2.2.2 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$_{opt}$
ClassType: Name TypeArguments$_{opt}$
InterfaceType: Name TypeArguments$_{opt}$
ArrayType: Type `[ ]`
TypeArguments: `<` *ActualTypeArgument*$^{+}_{,}$ `>`
ActualTypeArgument:
  ReferenceType
  Wildcard
Wildcard: `?` *WildcardBounds*$_{opt}$

WildcardBounds:
    extends *ReferenceType*
    super *ReferenceType*

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 §5 for the significance of *TypeArguments*.

2.3 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

$$\text{X: } \boxed{42}$$

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

2.4 Applying the model to Java

As modern languages in the Algol family go, Java is fairly simple[3]. 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.)

[3]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."

- 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).

- 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 3

Numbers

3.1 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.

3.1.1 Integral values and their literals

Syntax

IntegerLiteral: *IntegerNumeral* *IntegerTypeSuffix*$_{opt}$
IntegerNumeral: `0`
  PositiveDigit *DecimalDigit*$^{+}$
  `0` *OctalDigit*$^{+}$
  `0x` *HexDigit*$^{+}$
  `0X` *HexDigit*$^{+}$
PositiveOctalDigit: `1` | `2` | `3` | `4` | `5` | `6` | `7`
OctalDigit: `0` | *PositiveOctalDigit*
PositiveDigit: *PositiveOctalDigit* | `8` | `9`
DecimalDigit: `0` | *PositiveDigit*

Type	Modulus	Minimum	Maximum
`long`	2^{64}	-2^{63}	$2^{63}-1$
		(-9223372036854775808)	(9223372036854775807)
`int`	2^{32}	-2^{31}	$2^{31}-1$
		(-2147483648)	(2147483647)
`short`	2^{16}	-2^{15}	$2^{15}-1$
		(-32768)	(32767)
`byte`	2^{8}	-2^{7}	$2^{7}-1$
		(-128)	(127)
`char`	2^{16}	0	$2^{16}-1$
		0	(65535)

Table 3.1: Ranges of values of Java integral types. Values of a type represented with n bits are computed modulo 2^n (the "Modulus" column).

HexDigit:
  DecimalDigit
  `a` | `b` | `c` | `d` | `e` | `f`
  `A` | `B` | `C` | `D` | `E` | `F`
IntegerTypeSuffix: `l` | `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 3.1. As we will see in §3.1.2, 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; to get values of other types, one can use *casts.* The cast notation '(*T*) *V*' means "*V* converted to type *T*". Thus,

```
(short) 15  // is a short value
(byte) 15   // is a byte value.
```

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.

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 3.1, 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 §3.1.2. 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. 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)`

Syntax.

CharacterLiteral: `'` *SingleCharacter* `'`
`'` *EscapeSequence* `'`
SingleCharacter: Any Unicode character except CR, LF, `'` or `\`
EscapeSequence:
`\b` | `\t` | `\n` | `\f` | `\r` | `\"` | `\'` | `\\`
OctalEscape
OctalEscape: *ZeroToThree* $OctalDigit_{opt}$ $OctalDigit_{opt}$
ZeroToThree: `0` | `1` | `2` | `3`

Escape	Unicode	Meaning	Escape	Unicode	Meaning
`\b`	`\u0008`	Backspace (BS)	`\r`	`\u000d`	Carriage return (CR)
`\t`	`\u0009`	Horizontal tab (HT)	`\"`	`\u0022`	Double quote
`\n`	`\u000a`	linefeed (LF)	`\'`	`\u0027`	Single quote
`\f`	`\u000c`	Form feed (FF)	`\\`	`\u005c`	Backslash

Table 3.2: 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.

Here, 'LF' and 'CR' refer to the characters "linefeed" (also referred to as *newline*) and "carriage return."

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 3.2.

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'
'ÿ'          // Another way to write '\377' (if your
             // compiler supports it)
```

3.1.2 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) ≡ x
```

so that

```
x % y ≡ x - ((x / y) * y)
```

Working out a few examples:

```
5 % 3 ≡ 5 - ((5 / 3) * 3) ≡ 5 - (1*3) ≡ 2
-5 % 3 ≡ (-5) - (((-5) / 3) * 3) ≡ (-5) - ((-1)*3) ≡ -2
5 % (-3) ≡ 5 - ((5 / (-3)) * (-3)) ≡ 5 - ((-1)*(-3)) ≡ 2
(-5) % (-3) ≡ (-5) - (((-5) / (-3)) * (-3))
            ≡ (-5) - (1*(-3))
            ≡ -2
```

Division or remaindering by 0 throws an `ArithmeticException` (see **??**).

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 **int**s. 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 3.1. 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

$$527 \cdot 1000 + 600 = 527600 \equiv -16 \bmod 256;$$

or we can first convert all the numerals to `byte`s:

$$15 \cdot -24 + 88 = -272 \equiv -16 \bmod 256;$$

or we can convert the result of the multiplication first:

$$527000 + 600 \equiv 152 + 600 = 752 \equiv -16 \bmod 256.$$

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 `byte`s (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
```

Why this way? All these remainders seem rather tedious to us humans, but because of the way our machines represent integer quantities, they are quite easy for the hardware. Let's take the type `byte` as an example. Typical hardware represents a byte x as a number in the range 0–255 that is equivalent to x modulo 256, encoded as an 8-digit number in the binary number system (whose digits—called *bits*—are 0 and 1). Thus,

```
0    --> 00000000₂
1    --> 00000001₂
2    --> 00000010₂
5    --> 00000101₂
127  --> 01111111₂
-128 --> 10000000₂   (≡ 128 mod 256)
-1   --> 11111111₂   (≡ 255 mod 256)
```

As you can see, all the numbers whose top bit (representing 128) is 1 represent negative numbers; this bit is therefore called the *sign bit*. As it turns out, with this representation, taking the remainder modulo 256 is extremely easy. The largest number representable with eight bits is 255. The ninth bit position (100000000_2) represents 256 itself, and all higher bit positions represent multiples of 256. That is, every multiple of 256 has the form

$$b_0 \cdots b_n \underbrace{00000000}_{8},$$

which means that to compute a result modulo 256 in binary, one simply throws away all but the last eight bits.

Be careful in this notation about converting to a number format that has more bits. This may seem odd advice, since when converting (say) `byte`s (eight bits) to `int`s (32 bits), the value does not change. However, the `byte` representation of -3 is 253, or in binary 11111101_2, whereas the `int` representation of -3 is

$$11111111111111111111111111111101_2 = 4294967293.$$

In converting from a `byte` to an `int`, therefore, we duplicate the sign bit to fill in the extra bit positions (a process called *sign extension*). Why is this the right thing to do? Well, the negative quantity $-x$ as a `byte` is represented by $2^8 - x$, since $2^8 - x \equiv -x \bmod 2^8$. As an `int`, it is represented by $2^{32} - x$, and

$$2^{32} - x = (2^{32} - 2^8) + (2^8 - x) = 11111111111111111111111100000000_2 + (2^8 - x).$$

3.1.3 Manipulating bits

Syntax.

UnaryExpression: `~` *Operand*
BinaryExpression: Operand BinaryOperator Operand
BinaryOperator: `&` | `^` | `|` | `<<` | `>>` | `>>>`

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)* (0,0)	(0,1)	(1,0)	(1,1)
`&` (and)	0	0	0	1
`|` (or)	0	1	1	1
`^` (xor)	0	1	1	0

Operation	*Operand Bit* 0	1
`~` (not)	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 promoted to **int** or **long**, 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 == 2^30 - 2
                            |                 // 00111...1110
                            | (-y) >> 1 == -4
```

As you can see, even though these operators manipulate bits, whereas `int`s are supposed to be numbers, no conversions are necessary to "turn `int`s 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 111111111_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?]

3.2 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.

3.2.1 Floating-Point Literals

Syntax

RealLiteral:
- *DecimalDigit*$^+$ `.` *DecimalDigit** *Exponent*$_{opt}$ *FloatType*$_{opt}$
- `.` *DecimalDigit*$^+$ *Exponent*$_{opt}$ *FloatType*$_{opt}$
- *DecimalDigit*$^+$ *Exponent* *FloatType*$_{opt}$
- *DecimalDigit*$^+$ *Exponent*$_{opt}$ *FloatType*

Exponent:
- `e` *Sign*$_{opt}$ *DecimalDigit*$^+$
- `E` *Sign*$_{opt}$ *DecimalDigit*$^+$

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`$\pm N$' means $\times 10^{\pm N}$, and otherwise these literals are interpreted as ordinary numbers with decimal points. For example, `1.004e-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 `0.2f`, you get instead 0.20000000298....

Floating-point literals other than zero must be in the range

```
1.40239846e-45f to 3.40282347e+38f (float)
4.94065645841246544e-324 to 1.79769313486231570e+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**.

3.2.2 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) ≡ 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[1].

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 ± 0 by ± 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

$$0.1000000000000000055511151231257827021181583404541015625$$

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

$$0.12299999999999999822364316\ldots$$

[1]The handling of rounding to integer types is *not* the IEEE convention; Java inherited it from C and C++.

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*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 9×10^{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 significand 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, $\mathtt{x}\cdot\mathtt{y}$, by

$$\mathtt{x*y} = \mathtt{x}\cdot\mathtt{y}\cdot(1+\epsilon),\ \text{where } |\epsilon| \leq 2^{-53}.$$

and we say that ϵ 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 3.1 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^{th} approximation to $\sqrt{y}$. We could decide to stop when the error $|y - x_i^2|$ become small enough[2]. 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

[2]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.

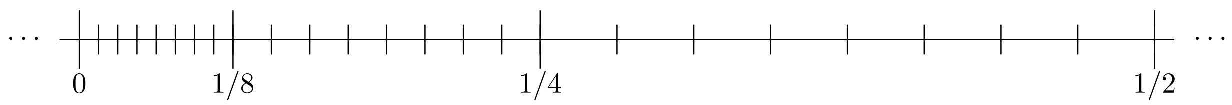


Figure 3.1: Non-negative 3-bit floating-point numbers near 0, showing how the spacing expands as the numbers get bigger. Each tick mark represents a floating-point number. Assume that the minimum exponent is -4, so that the most closely spaced tick marks are $2^{-6} = 1/64$ apart.

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 `int`s or `long`s. 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`.

Chapter 4

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 `String`s, 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` (§4.6), you can describe the format of textual input. On the other side of the coin, the construction of `String`s 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 (§4.2.1).

Input and output by a program requires additional abstractions (§4.4). 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).

4.1 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 **char**s 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.

4.1.1 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 4.1. 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 §3.1.1 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 (§4.2, S3.1.1) 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 `\u`*abcd* (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 δ = Ψ_1 - Ψ_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 §4.4, files are treated as streams of bytes, but the methods we programmers apply to read or write generally deal in **char**s. 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_1								
d_0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	^@	^A	^B	^C	^D	^E	^F	^G	^H	^I	^J	^K	^L	^M	^N	^O
1	^P	^Q	^R	^S	^T	^U	^V	^W	^X	^Y	^Z	^[	^\	^]	^^	^_
2	␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
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 4.1: ASCII codes (equivalent to the first 128 Unicode characters). The code for the character at line d_0 and column d_1 is 0xd_0d_1. The characters preceded by carats ^ are control characters, further described in Table 4.2; the notation means that the character ^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	^@	Null	DLE	^p	Data link escape
SOH	^a	Start of heading	DC1	^q	Device control 1
STX	^b	Start of text	DC2	^r	Device control 2
ETX	^c	End of text	DC3	^s	Device control 3
EOT	^d	End of transmission	DC4	^t	Device control 4
ENQ	^e	Enquiry	NAK	^u	Negative acknowledge
ACK	^f	Acknowledge	SYN	^v	Synchronize
BEL	^g	Bell	ETC	^w	End transmitted block
BS	^h	Backspace	CAN	^x	Cancel
HT	^i	Horizontal tab	EM	^y	End of medium
LF	^j	Line feed	SUB	^z	Substitute
VT	^k	Vertical tab	ESC	^[	Escape
FF	^l	Form feed	FS	^\	File separator
CR	^m	Carriage return	GS	^]	Group separator
SO	^n	Shift out	RS	^^	Record separator
SI	^o	Shift in	US	^_	Unit separator
			DEL	^?	Delete or rubout

Table 4.2: ASCII control characters and their original meanings.

4.1.2 The class Character

Besides its use as a wrapper for primitive characters (see §5.5), the class `Character` is stuffed with useful methods for handling **char**s. 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.

4.2 Strings

Pages 41–43 list the headers of the extensive set of operations available on `String`s. In addition, the core Java language provides some useful shorthand syntax for using some of these operations, as described in §4.2

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 `String`s makes certain uses of them a bit clumsy or slow, so there is another class, `StringBuffer`, whose individual characters *can* be changed (see §**??**).

4.2.1 Constructing Strings

You will almost never have to construct `String`s using **new** `String(···)`, so I have not bothered to show any in Figure 4.1. String literals (like `""` or `"Hello!"`) handle most constant strings.

The `valueOf` methods allow you to form `String`s 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 `Object`s (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 §4.2.1 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 4.1: 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 4.1, 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} + ... + s_{n-1} mod 2^{32} */
    public int hashCode();
    /** A String .equal to this and having the property that
     *      s0.equals (s1) iff s0.intern () == s1.intern (). */
    public String intern();
}
```

Figure 4.1, 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 `String`s, 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 ('`[`$\cdots$`]`') 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 4.3 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 §5.5), 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|"`

4.2.2 Aside: Variable-Length Parameter Lists

I sort of snuck a new feature past in the discussion of `format`. As shown in Figure 4.1, its definition begins

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

Those dots are actually written as shown, and mean "any number of parameters may follow here, each of which must be an `Object`." You may actually write something like this as the last parameter in any method definition. The type need not be `Object`. Although it looks like a major new mechanism, it's actually a pretty simple shorthand for something you could write otherwise. Let's take a full example with an actual implementation:

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 W.
‘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 (*fixed point,* as in 186000). If present, P gives the number of decimal places.
‘e’, ‘E’	floating point	format in scientific notation (as in 1.86e5). If present, P 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, P is the number of significant figures.
‘s’, ‘S’	general	convert to `String`, usually using `toString`. With ‘S’, convert to upper case first. If present, P 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 `Formattable` 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 4.3: Common format conversion characters and their meanings. See the text for the meaning of “category.”

```
/** 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. What is happening here is simply that the definition of the parameter with the ellipses (...), also known as a *varargs parameter*, is actually an array, as if the definition had been written

```
public static String makeList (String separator, String[] args) {
```

except that when the Java compiler sees that you have not given it an array (as in the first call to `makeList` above), it creates one for you, so that

```
makeList (" ", "The", "quick", "brown", "fox")
  is implicitly re-written to
makeList (" ", new String[] {"The", "quick", "brown", "fox"} )
```

The Scheme and Common Lisp languages have a similar feature.

In the case of `format`, something else is apparently also going on, because (as in some of our examples) we can supply arguments of type **int** or **double**, which are not `Objects`. We'll see how that works in §5.5.

4.2.3 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`.

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).

4.2.4 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.

4.2.5 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`.hashCode()==`S_1`.hashCode()` *if* S_0`.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;
}
```

4.3 StringBuilders

You can think of a `StringBuilder` as a *mutable* (modifiable) `String`[1]. 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)
```

[1]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 ();
    }
```

4.4 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 their 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(END0).  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 4.2: 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 4.2, 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 4.2, 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.

4.4.1 Input

The classes handling input in Java are loosely organized as follows:

- The abstract class `java.io.InputStream` represents a stream of **byte**s.
- 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 **char**s.
- 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 §4.6).

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 §4.6.

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* 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);
   }
   ...
```

4.4.2 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 **char**s 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 `Reader`s 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 `Reader`s and `InputStream`s.

4.4.3 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 **byte**s.
- 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, `PrintStream`s 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 **char**s.
- Concrete implementations of `Writer` include:

  `java.io.OutputStreamWriter` converts its stream of **char**s that it receives into a stream of bytes and gives it to an `OutputStream`.

`java.io.StringWriter` collects its stream of **char**s 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 **char**s.

`java.io.PrintWriter` is analogous to `PrintStream`.

4.4.4 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));
```

4.5 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 `String`s 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 "regexp" 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 4.4.

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]+")
```

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);
```

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, ... Any of the c_i may also have the form x-y, meaning "any single character $\geq x$ and $\leq y$."
`[^`$c_1c_2\cdots$`]`	Any single character *other than* c_1, c_2, ... 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).
`\s`	Any whitespace character (blank, tab, newline, etc.).
`\S`	Any non-whitespace character.
`\d`	Any digit.
`^`	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 *closure* 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 *either* R_0 or R_1.

Table 4.4: 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))*`'.

```
    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.

4.6 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 5

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 `Person`s 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[1]:

```
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.

5.1 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:*

[1]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 `Person`s 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[2]:

```
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.

5.2 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

[2]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`.

5.3 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*.

5.4 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 n = 1; n <= list.size (); i += 1)
      System.out.println (n + ". " + list.get (i));
}
```

(The '`+`' operation on `String`s 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 n = 1; n <= list.size (); i += 1)
      System.out.println (n + ". " + 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 n = 1; n <= list.size (); i += 1)
      System.out.println (n + ". " + list.get (i).name);
}
```

5.5 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 `Object`s: you can have lists of `Object`s and maps from `Object`s to `Object`s, but you can't use the same classes to get lists of **int**s. 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:

Primitive	$\Longrightarrow$	*Wrapper*	*Primitive*	$\Longrightarrow$	*Wrapper*
long	$\Longrightarrow$	`Long`	**char**	$\Longrightarrow$	`Character`
int	$\Longrightarrow$	`Integer`	**boolean**	$\Longrightarrow$	`Boolean`
short	$\Longrightarrow$	`Short`	**float**	$\Longrightarrow$	`Float`
byte	$\Longrightarrow$	`Byte`	**double**	$\Longrightarrow$	`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; `Integer`s 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.

5.6 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[3], 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 `Person`s. 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;

  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!

[3] 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.

Chapter 6

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 77.

Figure 6.1: The interface `java.lang.Runnable` and class `java.lang.Thread`. See also §6.1.

Class Thread *continued from page 76.*

```
    /** 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 · MILLIS + 10^-9 · 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 · MILLIS + 10^-9 · 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 6.2: Class`java.lang.Thread`, continued.

6.1 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 6.1–6.2 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.

6.2 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 ();
   }
}
```

6.3 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 preceding; 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.

6.3.1 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 §6.3.3.)

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.

6.3.2 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.

6.3.3 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:

```
                                tmp = X.z;
    while (X.z < 100) {         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`[1] 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.

6.4 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[2]. 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

[1]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.

[2]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 6.1–6.2, 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