Chapter 5  Expressions: Doing Things With Things

Chapter Overview

• How do I use the Things I have to get new (or other) Things?

This chapter and the next introduce the mechanics of executable code, the building blocks for individual sequences of instruction-following. The previous chapter's Things each come with a Type, which specifies how that Thing can interact. An expression is a piece of code that can be evaluated to yield a value and a type.

Simple expressions include literals -- Things that java literally understands as you write them -- and names, which stand in for the Things they refer to. More complex expressions are formed by combining other Things according to their types, or promised interactions.

To understand a complex expression, you must understand its parts (a basic form of "what goes inside") and how they are combined (a basic "how they interact"). Sometimes, you have to understand this without knowing all of the details of what's inside.

Sidebars in this chapter cover details of various Java operators, including casts

©1999 Lynn Andrea Stein. This chapter is excerpted from a draft of Interactive Programming In Java, a forthcoming textbook from Morgan Kaufmann Publishers. It is an element of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Permission is granted to copy and distribute this material for educational purposes only, provided that the following credit line is included: "©1999 Lynn Andrea Stein." In addition, if multiple copies are made, notification of this use must be sent to ipij@ai.mit.edu or ipij@mkp.com.
and coercion rules. In addition, supplementary reference charts are provided outlining the syntax and semantics of Java expressions.

### Objectives of this Chapter

1. To understand that an expression is a piece of Java code with a type and a value.
2. To become familiar with the rules of evaluation for basic Java expressions.
3. To learn how to understand complex expressions as combinations of simpler expressions.
4. To be able to evaluate both simple and complex expressions.

### 5.1 Simple Expressions

An expression is the simplest piece of Java code. An expression is a Thing, so it has both a value and a type. An instruction-follower—an execution of Java code—evaluates an expression to obtain its value, which will always be of the expression's type. There are many kinds of expressions, and each has its own rules of evaluation that determine what it means for an instruction-follower to evaluate that expression. Legitimate Java expressions include $2 + 2$, "Hi, there", and `this.out.writeOutput( this.in.readInput() )`. The last of these is an expression whose evaluation involves inter-object (and inter-entity) communication.

#### 5.1.1 Literals

The very simplest Java expression is a **literal**: an expression whose value is interpreted literally, such as `25` or `32e-65` or "How about that?". Java literals include the various kinds of numbers, characters, Strings, and booleans. For a more complete enumeration of literal expressions and rules regarding their **syntax** (i.e., how you write them), see the sidebar on Java Primitive Types, above.

Every expression has a value and a type, obtained by evaluating the expression. The value of a literal is its **prima facie** value, i.e., what it appears to be. The type of an expression is the type of its value. Integer literals are always of type `int` unless an explicit type suffix (`l`, `s`, or `b`) is included in the literal. Non-integral
numeric literals are always of type double unless explicitly specified to be of type float (using the f suffix).

### 5.1.2 Names

Names are also Java expressions. A name is only a legitimate expression once it has been declared, i.e., within its scope.¹ The value of that name is the value currently associated with it -- i.e., stored in the shoebox if it is a shoebox name, or labeled by it if it is a label name. The type of a name expression is always the type associated with that name at the time of its definition.²

For example, if we are within the scope of a declaration that says

```java
int myFavoriteNumber = 4;
```

and nothing has occurred to change the value associated with (stored in the shoebox called) `myFavoriteNumber`, then the value of the expression

```java
myFavoriteNumber
```

is 4 and its type is int. That is, the int 4 is the result of evaluating `myFavoriteNumber`.

### 5.2 Method Invocation

Method invocation is the primary way in which one object asks another to do something. It is the primary basis for inter-entity communication and interaction, because it is the main way in which objects talk to one another.

We have seen in previous chapters that objects are able to perform certain services. These service requests are called methods, and asking an object to do

---

¹ Strictly speaking, the area of text within which a name is legal is called its **scope**. The scope of a variable -- a name with no special properties beyond being a name -- begins at its declaration and extends to the end of the enclosing block. (See blocks, below.) Later, we will see three other kinds of names: classes, fields and parameters. Class names have scope throughout a program or package; they may be used anywhere. Field names have scope anywhere in their enclosing class, including textually prior to their declaration. Parameter names have scope throughout their method bodies only.

² Note that the type of a name expression is the declared type of the name rather than the type of the value associated with the name. That is, even where there is disagreement between the declared type of a name and its value, the type of a name expression is always its declared type.
something is called **method invocation**. In Java, a method invocation involves:

- An expression whose value is the object to whom the request is directed, followed by
- A period (or "dot"), followed by
- The name of the method to be invoked, followed by
- Parenteses, within which any information needed by the method request must be supplied.

An example method invocation might be

```
"a test string".toUpperCase()
```

This example consists of a String literal expression ("a test string") and a request to that object to perform its `toUpperCase()` method. A String's `toUpperCase()` method doesn't require any additional information, so the parentheses are empty. (They can't be omitted, though!) The value of a String's `toUpperCase()` method is a new String that resembles the old one, but contains no lower case letters. So the value of this expression is the same as the value of the literal expression "A TEST STRING".

Another example of method invocation is

```
Console.println("Hello")
```

This asks the object named by the name expression `Console` to print the line supplied to it. It requires that a String -- the line to be printed -- be supplied inside the parentheses. This is "necessary information" is called an **argument** to the method.

What is the value of this method invocation expression? `Console.println("Hello")` is a method invocation whose primary use, like that of assignment, is for its side effect, not its value. We use this method to make something appear on the user's screen. Good style dictates that we wouldn't use this expression inside any other expression. It turns out that many methods have no real return values, so (as we saw in the previous chapter) there's a special Java type for use on just such occasions. This type is called **void**. It is only used for method return types, and it means that the method doesn't return anything.

The evaluation rule for a method invocation expression is as follows:

1. Evaluate the object expression to determine whose method is to be
2. Evaluate any argument subexpressions.

3. Evaluate the method invocation by asking the object to perform the method using the information provided as arguments.

4. The value of the expression is the value returned by the method invocation. The type of the method invocation expression is the declared return type of the method invoked.

In order for step 3 to work, the object must know how to perform the method, i.e., it must have instructions that can be followed in order to produce the return value needed in step 4. We have already seen how an interface can describe an object's commitment to provide such behavior. We will see in the next chapters how this may be accomplished in detail.

From the perspective of the method invoker, however, the transition from step 3 to step 4 happens by magic (or by the good graces of the object whose method is invoked). The object offers the service of providing a particular method requiring certain arguments and returning a value of a particular type. For example, if we look at the documentation (or code) for String, we will see that it has a toUpperCase() method that requires no arguments and returns something of type String. The println method of Console requires a String as an argument, and println's return type is void. We will learn more about the methods that objects provide the chapters on Classes and Objects and Designing with Objects.

5.3 Combining Expressions

Since expressions are things -- with types and values -- expressions can be combined to build more complicated expressions. For example, the expression "serendipitous".toUpperCase() has the type String and the same value as the literal "SERENDIPITOUS". That is, you can use it anywhere that you could use the expression "SERENDIPITOUS". So, for example, you could get an adverbal form of this adjective by using "serendipitous".toUpperCase() + "LY", producing "SERENDIPITOUSLY", or extract the word "REND" using "serendipitous".toUpperCase().substring(2,5).

In general, since every expression has a type, you can use the expression wherever a value of that type would be appropriate. The exception to this rule about reuse of expressions is that some expressions are constant -- their value is
fixed -- while other expressions are not. Some contexts require a constant expression. In these cases, you cannot use a non-constant expression of the same type. (For example, "to"+"get"+"her" is a constant expression, but str+"ether" is (in general) not, even if str happens to have the value "tog".) There are a few places where Java requires a constant value. These will be noted when they arise.

The evaluation rule for a compound expression is essentially the same as the evaluation rules for the expressions that make it up: Evaluate the subexpressions that make up this expression, then combine the values of these subexpressions according to the evaluation rule for this expression. For example, when we evaluate "serendipitous".toUpperCase(), we are actually evaluating the simpler (literal) expression "serendipitous", then evaluating the method invocation expression involving "serendipitous"s toUpperCase() method. Similarly, str + "ether" evaluates the (name) expression str and the (literal) expression "ether", and then combine these values using the rules for + expressions, detailed below. In this case, str and "ether" are subexpressions of str + "ether". There are two additional details: 1) Evaluating the subexpressions may itself involve several evaluations, depending on how complex these expressions are and 2) it may not always be clear which operation should be performed first.

Method invocation, like other expressions, can be used to form increasingly complex expressions. For example, we can combine two method invocations we used above to cause the value of "A TEST STRING" to appear on the user's screen:

```java
Console.println( "a test string".toUpperCase() )
```

In this case, the value of the toUpperCase() invocation is used as an argument to println. We can also cascade other kinds of expressions, such as

"This is " + "a test string".toUpperCase()

or

```java
Console.readln().toUpperCase()
```

---

3 The expression str+"ether" would be constant if str were declared final, though. Names declared to be final cannot be assigned new values.
5.4 Assignments and Side-Effecting Expressions

Another kind of operator is assignment. We have already seen some simple assignments -- including some that were mixed with declarations and buried inside definitions. An assignment is actually a kind of expression. Its first operand -- the expression on the left-hand side -- must be a name or another expression that can refer to a shoebox or a label. In this context, and in this context only, the name expression refers to the shoebox or label, not to the particular value currently associated with the name.

Like all expressions, every assignment has a type and returns a value. The type of an assignment is the type of its left-hand side. The value of an assignment expression is the value assigned to the left-hand side. For example, the type of the expression

\[ \text{myNumber} = 4563129 \]

is int, because the type of 4563129 is int, and the value is 4563129 for the same reason.

Note that we must have declared myNumber before we get to this expression; and that this expression is legitimate if myNumber has type int, long, float, or double. Note, also, that if myNumber were already declared, we wouldn't want to declare it again. Every time that you declare a name, it creates a brand new shoebox or label with that name.

Although assignments are expressions in Java, they are not generally used for the resulting value. Instead, an assignment statement is generally used because it will cause the shoebox or label on its left-hand side to be associated with a new value. This effect is not a part of the value of the expression; instead, it happens "on the side" and is called a side effect. Assignment statements are among the most common expressions used for their side effects, but we will see several other expressions with important side effects in the remainder of this chapter.
5.5 Other Expressions that Use Objects

We have already seen method invocation, perhaps the most common object expression. In this section, we cover three additional expressions that use objects: field access, instance creation, and type membership. Each of these kinds of expressions will be discussed further when we explore how objects are actually created, beginning in the chapter on Classes and Objects.

5.5.1 Fields

In addition to methods, objects sometimes have fields: data members that behave as names. That is, fields are either shoeboxes or labels. Like methods, fields are also accessed using the dot syntax, but without following parentheses. A field access expression is essentially a name expression, though a more complex one than the simple names described above. The value of a field access expression is, as for a simple name, the value associated with the shoebox or label. So, for example, Math.PI is a double shoebox, belonging to an object called Math, containing a value approximating a real number whose most significant digits are 3.14159.

We can use field invocations in compound expressions, too. If myWindow is a Window with a getSize() method that returns a Dimension, `myWindow.getSize().height` first asks myWindow to perform its `getSize()` method, resulting in a particular Dimension object, then asks the Dimension object for its height field. This compound expression is the same as first creating a
name for the Dimension and assigning it the result of the method invocation:

```java
dimension mySize = myWindow.getSize();
```

and then asking the newly named Dimension mySize for its height field.

Because field access expressions are actually name expressions, they also have special behavior in the specific context of the target of an assignment statement. That is, you can assign to a field access expression just as you would to a simple name, and the field access expression behaves like the shoebox or label to which it refers. For example, if `height` is an `int` shoebox owned by `mySize`, the expression

```
mySize.height = mySize.height / 2
```

halves the value contained in the `height` shoebox of `mySize`, which might shrink `mySize` vertically by half.

### 5.5.2 Instance Creation

A second object-related expression is the `new` expression, used with a class name to create a new object. The details of this expression type are covered in the chapter on Classes and Objects; for now it is enough to recognize it. A `new` expression has three parts: the keyword `new`, the class name, and a (possibly empty) list of arguments, enclosed in parentheses. This description of how to write an expression is called its **syntax**, and we can abbreviate it as:

```
new ClassName ( argumentList )
```

The words in italics -- `ClassName` and `argumentList` -- are placeholders to indicate that you need to supply the details. The rest of the expression -- `new` and the parentheses -- are to be taken literally. For example,

```
new File ( "myData" )
```

creates a new `File` object with external (outside of Java) name `myData`. Like all other expressions, this one has a type -- `ClassName`, the kind of object created, in this case `File` -- and a value -- the new object created. The `new` expression is typically used inside an assignment or method invocation.

The rules of evaluation for creation expressions are similar to the rules of evaluation for method invocation. The return value is always a new instance of the type (or class) whose instance creation expression is invoked (in this case, `File`). The return type is always the type whose instance creation is invoked.
Instance creation is a side effecting expression (since it creates a new object).

### 5.5.3 Type Membership

There is one last operator that is useable only with objects. This is an operator called `instanceof`, which checks whether an object has (or can have) a certain type. It takes two operands:

```
anObjectExpression instanceof ObjectTypeName
```

The first operand, which precedes the keyword `instanceof`, can be any expression whose value is of any object (non-primitive) type. The second operand, which follows the keyword `instanceof`, must be the name of an object type. As we shall see in the next few chapters, this name may be the name of any class or any interface.

The `instanceof` operator is used to determine whether it is appropriate to treat its first operand according to the rules of the type named by its second operand. (For example, is it appropriate to cast the object to this type?) The value of an `instanceof` expression is a boolean, true if it is appropriate to treat the object according to this type, false otherwise. So, for example,

```
"a String" instanceof String
```

has the value true (because "a String" is a (literal) instance of the type String), while

```
new Object() instanceof String
```

has the value false (because the new Object created by the instance creation expression `new Object()` is not a String.

### 5.6 Complex Expressions on Primitive Types: Operations

Perhaps the most common kind of expression on primitive types is made up of two expressions combined with an `operator`. Java operators are described in the sidebar on Java Operators. They include most of the common arithmetic operators as well as facilities for comparisons, logical operations, and other useful functions. Of special note are `+` for String concatenation and unary `-` for negation.

Each operation takes arguments of specified types and produces a result with a particular value and type. For example, if \( x \) and \( y \) are both of type `int`, so is \( x + \)
The + operator can be used to combine any two numeric types. The two things combined with the operator are called the operands. In the expression $x + y$, + is the operator and $x$ and $y$ are the operands. Some operators take two operands. These are called binary operations. Other operators take only one operand; these are the unary operations. One operator -- ?: -- takes three operands.
Java operators include

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>^=</td>
<td>Bitwise XNOR</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Right shift</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left shift</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>? :</td>
<td>Ternary conditional</td>
</tr>
</tbody>
</table>

The operators in the first row are, respectively, addition, subtraction, multiplication, division, bitwise or, bitwise and, bitwise negation, modulus, left-shift, sign-extended right-shift, and zero-extended right-shift. Addition is also used for String concatenation when at least one of its arguments is a String. Subtraction can also be used as unary (one-argument) negation.

The operators in the second row combine their correlate in the first with an assignment operation. Thus `x += 2` is the same as `x = x + 2`; the difference is that the left-hand side of the combined operator is evaluated only once. The value of an operator assignment expression is the new value of the left-hand side; the type is the type of the left-hand side. All assignment expressions modify the name that is their left-hand side.

The third row above begins with six comparisons, each of which returns a boolean. The final comparison is not-equal. These are followed by logical negation, logical conjunction (and), and logical disjunction (or). Each of these takes boolean arguments, one in the case of negation, two in the case of conjunction and disjunction, and returns a boolean.

The final operators in the third row are autoincrement and autodecrement. These can be used as either prefix or postfix operators. Both `++x` and `x++` modify `x`, leaving it incremented. However, `++x` returns the incremented value of `x`, while `x++` returns the unincremented value. The `--` operator works similarly.

The final two operators are simple assignment (which works like the compound assignments, above) and the ternary (three-operand) expression conditional.

### 5.6.1 Arithmetic Operation Expressions

The operator `+` is an example of a kind of operator called an arithmetic operator. The rules for evaluation of the binary arithmetic operators `+`, `-`, `*`, `/`, and `%` are
simple: compute the appropriate mathematical function (addition, subtraction, multiplication, division, and modulus, respectively), preserving the types of the operands. As explained in the sidebar on Java Operators, `type op type has type type` for all of the basic arithmetic operations on most of the primitive type: For these arithmetic operators, if the types of the two operands are the same, the result -- the value of the complete expression -- will generally also be of that type. For example, evaluating `3 + 7` yields the `int` 10; `2.0 * 5.6` evaluates to produce 11.2, and -- perhaps surprisingly -- `5 / 2` evaluates to 2, not 2.5 (or 2.0).

Sometimes, an operator needs to treat one of its operands as though it were of a different type. For example, if you try to add `7.4` (a `double`) and `3` (an `int`), Java will automatically treat the `int 3` as though it were the equivalent double, `3.0`. This way, Java can add the two numbers using rules for adding two numbers of the same type. This kind of treating numbers -- or other things -- as though they had different type is called coercion. Coercion does not actually change the thing, it simply provides a different version (with a different type). For shoebox types, this version is essentially a copy. For label types, it is another "view" of the same object. Coercion is described more fully in the sidebar on Coercion and Casting.

Other arithmetic operators work in much the same way as `. Additional information on arithmetic expressions is summarized in the sidebar below. Note in particular that `/` (the division operator) obeys the same `type op type is type` rule. This means that `7 / 2` has type `int` (and the value 3). If you want a more precise answer -- 3.5 -- you can make sure that at least one operand is a floating point number: `7.0 / 2` has type `double`, as does `7 / 2.0`.

In addition to the `binary` (two-argument) arithmetic operators described above, Java includes a `unary` minus operator that takes one argument and negates it. So `-5` is an integer, while `-5` is an arithmetic expression that has value `-5` and type `int`. (Subtle, no?)
Arithmetic Expressions

Arithmetic expressions include the binary operators for addition (+), subtraction (−), multiplication (×), division (÷), and the modulus or remainder operation (%). In addition, there are two unary arithmetic operators, + and −.

Arithmetic operations work only with values of type int, long, float, or double. When a (unary or binary) arithmetic expression is invoked with a value of type short, byte, or char, Java automatically widens that operand to int (or to a wider type if the other operand so requires). For further details on widening, see the sidebar on Coercion and Casting.

When the operands of a binary arithmetic expression are of the same type, the complete expression also has that type, except that no binary arithmetic expression has type short, byte, or char. This is because operands of these types are automatically widened.

When the operands are of different types, Java will automatically widens one to the other.

The values of the expressions involving the binary operators +, −, ×, and ÷ are the sum, difference, product, and quotient of their (possibly widened) operands, respectively.

The value of $x \% y$ is the (appropriately widened) remainder when $x$ is divided by $y$.

The value of a unary − expression is the additive inverse of its (possibly widened) operand; a unary + expression has the value of its (possibly widened) operand.

5.6.2 Explicit Cast Expressions

If the numbers you wish to divide -- or otherwise combine -- are not literals, you can still change their types using an explicit cast expression (as described in the sidebar on Coercion and Casting). Like coercion, this gives you a view of the thing cast as a different type. It is accomplished by putting the name of the type that you wish the thing to have in parentheses before the (expression representing the) thing. For example, if myInt is an int-sized shoebox holding the value 3,
(long) myInt is a view of 3 as a long and (double) myInt is an expression with the same type and value as the literal expression 3.0. Throughout this, myInt itself remains an int-sized shoebox holding the value 3.

Evaluating a cast expression yields the value of the cast operand (in this case, myInt), but with the type of the explicit cast (in this case, long). A cast expression does not alter its operand in any way; it simply yields a new view of an existing value with a different type. Some casts are straightforward and appropriate; some risk losing information; and most are simply not allowed. For example, in Java you cannot cast an int to boolean. Casts are also allowed from one object type to another under certain circumstances. See the sidebar on Coercion and Casting for further details.
Coercion and Casting

Sometimes things don't have the types we might wish. Coercion is the process of viewing a thing as though it had a different type. Coercion does not change the thing itself; it merely provides a different view.

Java only makes certain automatic -- implicit -- coercions. For example, Java knows how to make `byte` into `short`, `short` into `int`, `int` into `long`, `long` into `float`, and `float` into `double`. This works because each type spans at least the magnitude range of the ones appearing before it in the list. (A few of these coercions-- such as `long` to `float` -- may lose precision.) These coercions -- which are, in general, information-preserving -- are called **widening**. We will see in the chapter on Objects and Classes that there are also widening coercions on reference types.

Coercions in the opposite direction are called **narrowing**. Java does not generally perform narrowing coercions automatically. For example, Java cannot automatically convert an arbitrary `int` to a `short`, because the `int` might contain too much information to fit into a `short`. The number 60000 is a perfectly legitimate value for an `int`, but not for a `short`. There is no mapping from `ints` to `shorts` that accurately captures the magnitude information in each possible `int`. A coercion of this kind -- such as `int` to `short`-- which may not preserve all of the information in the original object, is called **lossy**.

Sometimes, you need to change the type of an object when Java will not do so automatically. This is accomplished by means of an **explicit cast expression**. The syntax of a cast expression is

```
(type-name) expression to be cast
```

For example, if `myInt` is a name of type `int` with value 7 (e.g., `int myInt = 7;`), then

```
(long) myInt
```
is an expression with type long and value 7. (Note that myInt still has type int. Casting, like implicit coercion, does not actually modify the castee.)

Explicit coercion allows both widening and narrowing coercions: you can cast an int to long, as in the example above, or to short -- a cast that may lose information. Certain casts may be illegal and will cause (compile- or run-time) errors or exceptions.

5.6.3 Comparator Expressions

Not all operators are arithmetic. There is a set of boolean-yielding operators, sometimes called comparators, that operate on numeric types. These include <, <=, ==, etc. (See the sidebar on Java Operators for a complete list.) These take two numbers, coerce appropriately, and then return a boolean indicating whether the relationship holds of the two numbers in the order specified. For example, 6 > 3.0 is true, but 5 <= 3 is false. Beware: == tests for equality; = is the assignment operator (see below).

Equality testing -- the operators == and != -- are not restricted to numeric types. For any type, these operators combine two expressions of the same type, returning true only of both operands are the same. When are two operands the same? For primitive types, values are the same whenever they "look" the same, i.e., when their values are indistinguishable. For object types, values are the same exactly when the two expressions refer to the same object. It is not sufficient for two objects to look alike (as in the case of identical twins); they must actually be the same object, so that modifications to one will necessarily be reflected in the other. (This is like giving one twin an haircut as we did in the chapter on Things, Types, and Names.)

Evaluating one of these expressions is much like evaluating an arithmetic expression. The values of the operands are compared using a rule specific to the operator -- such as > or <= -- and the resulting boolean value is the value of the expression.

5.6.4 Logical Operator Expressions

Another set of operators combines booleans directly. These include && (conjunction, or "and") and || (disjunction, or "or"). For example, the
expression `true || false` is `true`. While this is not very interesting by itself, these boolean operators can be used with names (of type `boolean`, of course) or in complex expressions to great effect. For example, `rainy || snowy` might be a reasonable way to express bad weather; it will (presumably) have the value `true` exactly when it is precipitating. There is also a unary boolean negation operator: `!`. The Java fragment

```java
!(rainy || snowy || overcast)
```

might be a good expression for sunshine.

The rule for evaluating negation is simply to invert the boolean value of its operand. The rules for evaluating conjunction and disjunction are a bit more complex. First, the left operand is evaluated. If the value of the expression can be determined at this point (i.e., if the first operand to a conjunction is false or the first operand of a disjunction is true), evaluation terminates with this value. Otherwise, the second operand is evaluated and the resulting value computed. The type of each of these expressions is boolean.

These odd-seeming rules are actually quite useful. You can exploit them to insert tests. For example, you might want to compute whether `(x / y) > z`, but it might be the case that `y` is 0. By testing whether `(y == 0) || ((x / y) > z)`, you can eliminate the potential divide-by-zero error. (If `y` is 0, the first operand to the disjunction -- `(y == 0)` -- will be true, so evaluation will stop and the value of the whole will be true. (A comparable formula can be written to return false if either `y` is 0 or `(x / y) > z`.)

### 5.7 Parenthetical Expressions and Precedence

A parenthetical expression is simply an expression wrapped in a pair of parentheses. The value of a parenthetical expression is the value of its content expression, i.e., the value of the expression between the `(` and `)`. The type of a parenthetical expression is the same as the type of the expression between the parentheses. Parenthetical expressions are extremely useful when combining expressions. For example, in the previous section, we mentioned that

"I have " + x + 3 + " monkeys"

might yield 63 monkeys. We could fix this by rewriting the expression as

"I have " + (x + 3) + " monkeys"
This isolates \( x + 3 \) as a separate expression, making the + in \( x + 3 \) behave like addition, not String concatenation.

Note that, in giving the evaluation rules for expressions, white space doesn't matter -- \( x >= 2 + 3 \) is identical to \( x >= 2 + 3 \) -- but punctuation does. For example, \( 2 + 3 * 2 \) doesn't have the same value as \( 5 * 2 \). We can use parentheses to fix this, though: \( (2 + 3) * 2 \) is 10 again. In this case, parentheses change the order of evaluation of subexpressions (or, equivalently, how the expression is divided into subexpressions.) In the case of \( 2 + 3 * 2 \), if you evaluate the + first, then the *, you get \( 5 * 2 \), while if you evaluate the * first, you get \( 2 + 6 \).

How do you know which way an expression will be evaluated? In these situations, where one order of operation would produce a different answer from another, we fall back on the rules of precedence of expression evaluation. In Java, just as in traditional mathematics, * and / take precedence over + and -, so \( 2 + 3 * 2 \) really is 8. (Another way of saying this is that the * is more powerful than the +, so the * grabs the 3 and combines it with the 2 before the + has a chance to do anything. This is what we mean when we say that * has higher precedence than +: it claims its operands first.)

A full listing of the order of precedence in Java is included in the sidebar on Java Operator Precedence. Parentheses have higher precedence than anything else, so it is always a good idea to use parentheses liberally to punctuate your expressions. This makes it far easier for someone to read your code as well.
Java Operator Precedence

Expressions with multiple subexpressions are evaluated according to the rules of Java precedence. The following chart gives the rules for order of evaluation of Java expressions, with the expression types listed higher having higher priority, i.e., being evaluated first.

Operators in the table below are grouped by equivalent precedence. Within these groups, order of evaluation of an expression is from left to right in that expression.

Since an expression cannot be evaluated until its subexpressions have been, precedence determines the extent of operands to each operator, i.e., what the operand subexpressions of an operator are.

<table>
<thead>
<tr>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --, +, -, ~, !, explicit cast</td>
</tr>
<tr>
<td>*, /, %</td>
</tr>
<tr>
<td>+, -</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, instanceof</td>
</tr>
<tr>
<td>==, !=</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>? :</td>
</tr>
<tr>
<td>= and all compound assignments</td>
</tr>
</tbody>
</table>
Other Assignment Operators

Compound Assignment

Java has several variants on the simple assignment statement. If we have already declared total as an int, we can say:

```
total = 6
```
or

```
total = total + 1
```
(The second uses the fact that total + 1 is an expression with type int and value one greater than total to form an assignment expression whose second operand is an arithmetic expression.) This last expression -- adding to a name -- is pretty common, and so it has a convenient shorthand:

```
total += 1
```
The += operator is one of a class of **compound assignment operators**. It works by computing the value of its first operand, then adding its second operand to that value and assigning the result to the name represented by the first operand. In other words, the expression above is exactly the same as saying `total = total + 1`. This kind of compound assignment can be used with any number -- or other appropriate expression -- as the second operand, of course. There are also other compound assignment operators in Java, including -=, *=, /=, and %=, and the += operator works for both numeric addition and String concatenation. Like their longhand forms -- the simple assignment equivalents -- these expressions have type and value of their left-hand side (after the assignment).

AutoIncrement and AutoDecrement

There is another family of side-effecting operators that are related to assignment. These operators are **autoincrement** and **autodecrement**. The postfix autoincrement expression

```
total++
```
is similar to `total = total + 1` (or `total += 1`), but it has the value of total before the assignment. The **prefix** autoincrement expression

```
++total
```
also adds one to total, but has the value of total after the assignment. (Remember: \( ++\text{var} \) first increments, then produces a value; \( \text{var}++ \) produces the value first.) The two (prefix and postfix) autodecrement operators work similarly.

### Chapter Summary

- Every expression has both a type and a value.
- Simple expressions include literals and names.
  - A literal has its apparent type and value.
  - A name has its declared type and assigned value.
- Operator expressions combine or produce modifications of simpler expressions.
  - Arithmetic operators compute mathematical functions; the type of an arithmetic operation expression is typically the wider of its operand types.
  - Logical operators compute binary logical functions; the type of a logical operation expression is \textit{boolean}.
  - Explicit cast expressions have the type of the cast operation and the same value as the cast operand.
  - None of the above expressions actually modifies any of its operands. However, autoincrement, autodecrement, and the shift operators do modify their operands.
- Assignment expressions are generally used for their effects -- modifying the value associated with a (shoebox or label) name -- but, as expressions, also have type and value. The value of an assignment expression is the value assigned; the type is the type of the value assigned.
- Several kinds of expressions operate on objects:
  - A method invocation expression has the type and value returned by
the method. Methods may be side-effecting.

- A field access expression is like an ordinary name expression: its type is the field's declared type and its value is the field's current assigned value, except in the context of assignment expressions.

- A constructor expression's value is a brand new object whose type is the type with which the constructor expression is invoked.

---

**Exercises**

1. In Java, every expression has a type. Assume that the following declarations apply:

```java
int i, j, c;
double d;
short s;
long l;
float f;
boolean b;
```

For each expression below, if it is syntactically legal Java, indicate its type (not its value). If it is not syntactically valid, indicate why.

1. 6
2. 24L
3. +3.5
4. 3.5f
5. 2e-16
6. -25b
7. i
8. i+3
9. i+3.0
10. i+s
11. \( l+d \)
12. \( f+s \)
13. \( i / 0 \)
14. \( 4 \times 3.2 \)
15. \( i = 0 \)
16. \( i == 0 \)
17. \( b = 0 \)
18. \( b == 0 \)
19. '\c'
20. "An expression in double-quotes"
21. "An expression in double-quotes" + "another one"
22. "6" + 3
23. \(!b\)
24. \(!i\)
25. \(b || true\)
26. \(i += s\)
27. \(s += i\)
28. \(i += f\)
29. \(l = i = s\)
30. \(i = l += s\)
31. \(l++\)
32. \((long) s\)
33. \(s\)
34. \((short) l\)
35. 1

2. Give examples of three expressions with side effects.

3. What is the value of each of the following expressions? Which ones produce errors in evaluation? You may wish to consult the chart on operator precedence. Assume that i is an already-defined name for an int and that b is a boolean.

   1. 2.0 + 3.5 * 7
   2. ("top " + "to " + "bottom ").toUpperCase()
   3. "the answer is " + 6 * 7
   4. 4 + 6 + " is " + 10
   5. i > 0 && i < 100
   6. b = i < 0
   7. ! (i == 0) && 100 / i

4. Give examples of each of the following:

   1. An expression whose type is int and whose value is more than a previously defined int, x.
   2. An expression whose type is boolean and whose value is true when x is between 5 and 15.
   3. An expression whose type is double and whose value is half of x's, where x is the aforementioned int.
   4. An expression whose type is long and whose value is the remainder when x is divided by 7.
   5. An expression whose type is boolean and whose value is the opposite of a previously defined boolean, b.
   6. An expression whose type is boolean and whose value is true exactly when the int x is evenly divisible by 5.
   7. An expression whose type is String and whose value is read from the user's keyboard.