tml,v 1.1 2003/06/11 19:39:26 jon Exp $

Introduction to Interactive Programming

By Lynn Andrea Stein

A Rethinking CS101 Project

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. Chapter 3'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 Thing with 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 that 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.

This chapter introduces several new kinds of expressions. Instance creation expressions, field access, and type membership expressions all involve objects. Most operations involve primitive types. And some other expressions -- assignment, equality testing, and explicit casting -- can be used with either objects or primitive Things. For each of these kinds of expressions, this chapter explains its syntax as well as the type and value of the resulting Thing.

Sidebars in this chapter cover details of various Java operators, including casts and coercion rules. A separate supplementary reference chart -- Java Charts/Expressions -- summarizes the syntax and semantics of Java expressions.

Objectives of the 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, giving the resulting type and value.

5.1 Evaluating an Expression Produces a Thing

In the preceding chapters, we have learned that Things have types as well as values. We have seen how a type tells you what an instance of that type can do. We have also seen three different ways to get our hands on a Thing:

- A literal, like 6, is a Thing that Java understands directly. Rules for interpreting literals (from the sidebar on Java Primitive Types) specify the literal's type.
- A name, like theClock, refers to the Thing to which the name is bound. A name is declared to have a specific type.
- A request, like theClock.getTime(), returns the Thing that is the (return) value provided by theClock's getTime() service. A method signature specifies the type of Thing that a service returns.

Each of these ways of obtaining a Thing is an example of an ***expression***, a piece of code that can be evaluated to produce a value (or Thing) of a particular type. To ***evaluate*** an expression simply means to calculate the value of the expression. In a computer program, every value is produced by an expression. This chapter investigates many of the different kinds of expressions found in Java.

An expression is the simplest piece of Java code. An instruction-follower--an execution of Java code--evaluates the 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. Each kind of expression is distinguished by the way that its value and type are produced, and these -- together with the syntax of the expression -- are the important features of expressions that you will need to learn.

In addition to the three kinds of expressions mentioned above--literals, names, and service requests--there are expressions that do arithmetic, assign values to names, and create or compare objects. 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.

In this chapter, we will examine many different kinds of expressions and learn what values and types they produce. We will begin with the kinds of expressions that we have already seen.

5.1.1 A Literal Is What It Looks Like

Expressions are ways to get your hands on Things. A simple way to get your hand on a Thing is to spell out the Thing that you want. So 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 in chapter 3.

are the `int`s, `double`s, `char`s, `boolean`s, or `String`s that they appear to be.

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 `long` type suffix (l or L) 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 or F suffix).

Of course, not all Java Things can be spelled out explicitly. In fact, except for `String` objects, only Things with primitive types can be literals.

5.1.2 A Name Has An Associated Value (and Type)

Another simple way to get your hand on a Thing is to have it around in the first place. Names -- labels for objects, dials for primitives -- are also Java expressions. A name is only a legitimate expression within its legitimate lifetime, i.e., within its scope. For present purposes, it suffices to think of a scope as beginning when the name is declared. [[Footnote: Strictly speaking, 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 the section on **blocks** in the chapter on Statements.) 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.]]

has its declared type and most recently assigned value.

The value of a name is the value currently associated with it, i.e., labeled by it, if it is a label name, or recorded on the dial, if it is a dial name. The type of a name expression is always the type associated with that name at the time of its definition. [[Footnote: 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.]]

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

```
int myFavoriteNumber = 4;
```

and nothing has occurred to change the value associated with (recorded on the dial called) `myFavoriteNumber`, then the value of the expression

```
myFavoriteNumber
```

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

5.1.3 Method Invocation Asks an Object to Do Something

Literals and names are simple Java expressions. In each case, a single Thing is involved. We have also seen expressions in which an object is asked to do something or to provide us with a value. In the previous chapter, we learned that the way that an object performs the service is called a method. This kind of expression -- asking an object to provide a service -- is formally called ***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.

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
- Parentheses, within which any information needed by the method request must be supplied.

An example method invocation might be

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

ꞁf method invocation:

. method(arglist)

ype and value as declared by the object's method.

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!) This invocation matches the method signature of the `String toUpperCase()` method as described in the Selected String Behavior sidebar of Chapter 3.

The value of a String's `toUpperCase()` method is a new String that resembles the old one, but contains no lower case letters. Its type is the return type declared in the `toUpperCase()` method signature: `String` So the value and type of this expression are the same as the value and type 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. It corresponds to the parameter specified by `Console`'s `println` method signature.

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.

Method invocation differs from literals and names in that method invocation expressions can *contain other* expressions. In both of these examples -- `"a test string".toUpperCase()` and `Console.println( "Hello" )` -- expressions such as `"a test string"`, `"Hello"`, and even `Console` had to be evaluated before the method invocation expression itself could be evaluated. This makes method invocation a ***compound*** expression: one that can contain other expressions within itself.

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

1. Evaluate the object expression to determine whose method is to be invoked.
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 commitment 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 in the chapters on Classes and Objects and Designing with Objects.

5.2 Combining Expressions

Java Notes

String Concatenation

It is often useful to be able to combine two Strings by gluin them together. This is called ***concatenation***. For example, concatenating the Strings "silvers" and "word" results in the String "silversword". Because this is such a common operation, Java provides a shorthand for String concatenation. In Java, you can concatenate two Strings by writing a + between them:

```
"silvers" + "word"
```

is the same Thing as `"silversword"`. This is actually a form of operator expression.

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 adverbial form of this adjective by using `"serendipitous".toUpperCase() + "LY"`, producing the same Thing as `"SERENDIPITOUSLY"` (see the sidebar on String Concatenation), or extract the word "REND" using `"serendipitous".toUpperCase().substring(2,6)` (see the Selected String Behaviors sidebar in Chapter 3).

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. A few specific Java 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"`. [[Footnote: The expression `str+"ether"` *would* be constant if `str` were declared `final`, though. Names declared to be `final` cannot be assigned new values.]]) 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:

```
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

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

5.3 Assignments and Side-Effecting Expressions

Literals, names, and most method invocations are used to produce Things: values with types. However, some expressions are evaluated not for the Thing that result but for some other effect that happens when the expression is evaluated. This thing-that-happens is called a ***side effect***. We have actually seen examples of this already: an assignment is a prototypical side-effecting expression.

Style Notes

Don't Embed Side-Effecting Expressions

When you use a side-effecting expression, it is best if this expression is not a subexpression of any other expression. So, for example, while assignments--as expressions--*can* be used inside oth expressions, it is generally considered bad style to do so. Embeddin side-effecting expressions inside other expressions can make the logic of your code very difficult to follow. Side effects are also important and often difficult to catch. By highlighting the side effecting expression by making it the outermost expression, you ar increasing the likelihood that it will be read and understood.

In previous chapters, we have seen some simple assignments including some that were mixed with declarations and buried inside definitions. In its simplest form, an assignment involves a name and an expression whose value is to be assigned to that name:

```
aString = "Something to say...."
```

ıent is used for its side effect, *not* its type and value.

The point of the assignment expression is to associate the name -- `aString`, in this case -- with the value of the Thing after the `=`. It is this side effect -- the binding of a name to a value -- that is generally why one uses an assignment statement.

In its most general form, an assignment expression is

```
name = someExpression
```

The right-hand side of the assignment expression -- what comes after the `=` -- can be any expression. The ***left-hand side*** of the assignment expression -- what comes before the `=` -- *must* be a name or another expression that can refer to a label or a dial. In this context, and in this context *only*, the name expression refers to the label or dial itself, *not* to the particular value currently associated with the name. Of course, the type of the name on the left-hand side must be compatible with the type resulting from the evaluation of the expression on the right-hand side.

[[Footnote: Odd as it may seem, left-hand side -- or its shorthand, lvalue -- is actually a technical computer science term. It refers to the name in an assignment expression, which is treated as a location -- a label or dial -- rather than as a true sub-expression of the assignment. The term lvalue is used even when the programming language in which the assignment is written does not position the name on the left-hand side!]]

Although assignment is generally used for its side effect, it *is* an expression. This means that evaluation of an assignment must produce a Thing -- a value with a type. 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, i.e, the value produced by evaluating its right-hand side.

For example, assuming that `aString` has previously been declared to be a `String`, the type of the expression

```
aString = "Something to say...."
```

is `String` because that is the type associated with the name `aString`. The value of the expression is `"Something to say...."` because that is the value assigned to `aString`.

An assignment is a side-effecting expression: although its evaluation produces a type and a value, it has another effect that is often -- if not always -- more important. Some method invocations -- service requests -- are similarly side-effecting. For example,

```
Console.println( "^*&$%" )
```

is a side-effecting request that causes something to be printed on the user's computer screen. Side-effecting methods often -- though not always -- have return type `void` -- no return value. This helps to make it clear that such a method is to be used for its side effect, since it does not provide a useful return value. All `void` methods should be side-effecting -- otherwise they do nothing at all! -- but not all side-effecting methods are necessarily `void` methods.

Assignment statements and `void` methods are among the most common expressions used for their side effects. We will see several other expressions with important side effects in the remainder of this chapter. In the next chapter, we will explore the use to which such side-effecting expressions can be put.

5.4 Other Expressions that Use Objects

Some kinds of expressions work only with objects. We have already seen how to request that an object perform a service through 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. In this chapter, it is sufficient to recognize these kinds of expressions and to understand their associated types and values.

5.4.1 Fields

We have seen that an object may provide services in the form of methods. Some objects also provide data in the form of a name. For example, the object called `Math` provides a `double` (dial) called `PI`, which has (is set to) a value that is a little more than `3.14159`.

Like a method, a field is accessed using the dot syntax, but without following parentheses. So, for example,

```
Math.PI
```

is the `double` dial belonging to the object called `Math`, with a value approximating a real number whose most significant digits are `3.14159`.

cess is

. method

es like a name: declared type and most recently assigned value.

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 label or dial. The type of a field access expression is -- also like a simple name -- the field's declared type.

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:

```
Dimension mySize = myWindow.getSize();
```

and then asking the newly named `Dimension` object, called `mySize`, for its `height` field.

Because field access expressions are actually name expressions, they also have special behavior on the left-hand side 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 label or dial to which it refers. For example, if height is an `int` dial owned by `mySize`, the expression

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

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

5.4.2 Instance Creation

In the previous chapter, we looked at some object types called interfaces. An interface specifies an object's contract, but not its implementation. In chapter 7, we will learn about another object type, called a class. A class is an object type that has an associated implementation. A particular object -- an instance -- is often manipulated using an interface in order to separate the object's user from its implementation details. But when you want to create a new instance, you need to have an implementation as well as a contract. This means that you need a class.

Instance creation is a kind of expression that uses a class to create a new instance of that class. The details of this expression type are covered in the chapter on Classes and Objects; for now it is enough to recognize it.

An instance creation expression has three parts: the keyword `new`, the class (type) 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 )
```

f instance creation:

ssName(arglist)

***ClassName*; value is a new instance.**

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 instance creation expression is typically used inside an assignment or method invocation.

The rules of evaluation for instance 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).

5.4.3 Type Membership

There is one last operator that is usable only with objects. This is an operator called `instanceof`, which checks whether an object has (or can have) a certain type. For example, is the Thing in my pocket a `String`?

When would an object of one type be an instance of another? There are at least three different ways that this could happen, all of which are covered in more detail elsewhere in this book. First, one type may be a specialization of another. For example, a collie is a kind of a dog, so an object can simultaneously be a collie and a dog. Second, one type-contract (interface) can have multiple implementations (classes), like a telephone that may use wires or cellular transmission or satellite to send its signal. A cell phone, land line, and satellite telephone might each implement the `Telephone` interface but each have its own specific type (e.g. `CellularTelephone`). Finally, one (instance) object may implement many interface contracts: A Person may be a Parent, an Employee, and a SoccerGoalie. A daughter, an employer, and a soccerForward would each access this object through a different interface. In each of these cases, it is reasonable to ask, "Is this Dog (or Telephone or Person) a Collie (or CellularTelephone or SoccerGoalie)?"

A type membership expression looks like this:

```
anObjectExpression instanceof ObjectTypeName
```

For example,

```
myDog instanceof Collie

aTelephone instanceof CellularTelephone

mobyDick.author() instanceof SoccerGoalie
```

f type membership:
`instanceof` *`Type`*
s to a boolean, either `true` or `false`.

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. 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.

5.5 Expressions Involving Objects or Primitives

Method invocation and field access, instance creation and type membership are all expressions that involve objects. These kinds of expressions cannot be used with primitive Things, although -- except for instance creation -- their resulting values may have primitive types. For example, a method must be invoked on an object but may require an argument that is a `char` or return an `int`. A field belongs to an object, but may have type `double`. And `instanceof` expressions always return `boolean` values.

In contrast, literals -- except for `String` literals -- are always primitives. In the next section (5.6), we will look at other kinds of expressions that operate only on primitives and not on objects. These include various arithmetic and logical operations as well as numeric and logical comparators. All of these operations take advantage of the known structure of the primitive types to provide basic calculations.

There are a few kinds of expressions that operate equally on either primitives or object types. These expressions are the focus of this section. We have already seen assignment; in this section we will also introduce explicit cast expressions and equality testing.

Assignment expressions involve a name on the left-hand side and an expression of any type suitable for binding to that name on the right-hand side. This means that if the name on the left-hand side is of an object type -- a label name -- the right-hand side must be an expression with an (appropriate) object type; if the name on the left-hand side is a primitive -- dial -- name, the expression on the right-hand side must have a (corresponding) primitive type. The use of an assignment expression is identical whether it is operating over object or primitive types, although the mechanics of assigning to a label name differ from the mechanics of assigning to a dial name. (Two label names may be stuck on one object, while dial assignment involves copying the value of one dial onto another. These distinctions are described in detail in Chapter 3: Things, Types and Names.)

There is also one kind of expression that is primarily used with primitive Things, but that has one object application. This is `+`, whose use as the arithmetic addition operator is described below. The sidebar on String concatenation, above, explains how `+` can be used to glue two Things together. In this case, it is important to distinguish the use of `+` as addition from `+` as String concatenation. They are really two separate operations that simply happen to be spelled in the same way.

[[Footnote: There is a name for this phenomenon, the use of one operation for multiple purposes. It is called ***operator overloading***. This is very much like method overloading -- described briefly in Chapter 4: Interfaces and in somewhat more detail in later chapters. -- except that methods always belong to objects, not primitive Things. In some programming languages, the programmer can overload operators, giving them additional behavior. Java does not allow the programmer to overload operators, but a few operators -- like `+` -- have multiple built-in meanings.]]

5.5.1 Casting and Coercion: Changing the Type of a Thing

Sometimes Things don't have the types we might wish. For example, we might have the `int 3` when we really want the `double 3.0`, or a Dog who is really a Collie that can herd sheep. ***Coercion*** is the process of viewing a Thing of one type as though it had a different type: for example, treating an `int` (like `3`) as though it were a `double` (`3.0`) or a Dog as

Java Notes

Coercion and Casting in Java

Generally, coercion happens automatically whenever it is information-preserving.

Java only makes certain automatic -- implicit -- coercions. For example, Java knows how to make `byte` into `short`, `short` intc `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-- sucl 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 Inheritance 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 though it were a Collie. Coercion does not change the Thing; it merely provides a different view.

5.5.1.1 Widening and Narrowing

Some coercions, such as treating an `int` or a `float` as a `double` or a Collie as a generic Dog, are information-preserving. That is, every `int` (and every `float`) has an accurate representation as a `double`. These are called ***widening*** coercions, and Java can do them automatically when types demand it. For example, the (stylistically ugly) definition

```
double d = 3;
```

implicitly coerces the `int 3` into the `double 3.0` before making the assignment to `d`. Similarly, if you've made the appropriate definitions, Java can figure out that it's OK to treat a CellTelephone as a Telephone or a Collie as a Dog.

For example, Java cannot automatically convert an arbitrary `int` to `short`, because the `int` might contain too much information to fit into a `short`. The number 60000 is a perfectly legitimate value for a `int`, but not for a `short`. There is no mapping from `int`s to `short`s that accurately captures the magnitude information in each possibl `int`. A coercion of this kind -- such as `int` to `short`-- which may nc preserve all of the information in the original object, is called ***loss*** [[Footnote: There is one instance in which Java performs a narrowing but non-lossy coercion automatically. This is in the case of a sufficiently small `int` constant assigned to a narrower integer type. This allows literals-- which would otherwise have type int -- tc be assigned to names with byte and `short` type: `short smallNumbe = 32;`]]

Explicit casting 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. Casts on object types may be made only to other types that are legal types of the object being cast. [[Footnote: An object *x* may be cast to type *T* iff `x instanceof T` is true.]] Certain casts may be illegal and will cause (compile- or run-time) errors or exceptions.

Other coercions, such as treating the `double 3.0` as though it were an `int` or a Dog as a Collie, are not necessarily information-preserving. (The `double 3.001` might also be treated as the `int 3`, so this type transformation in general loses information.) You may know that a particular Dog is really a Collie, but Java won't automatically treat it as such since some Dog might not be a Collie. These coercions, called ***narrowing***, are not performed by Java automatically. Instead, you need to indicate a narrowing conversion to Java by using an explicit cast expression as described below.

Coercion is a way of looking at a Thing; it does not actually change the Thing itself. A coercion simply provides a different version (with a different type). For dial types, this version is essentially a copy. For label types, it is another "view" of the same object.

Java's specific treatment of coercion is described more fully in the sidebar on Coercion and Casting in Java.

5.5.1.2 Explicit Cast Expressions

Sometimes, you need to change the type of Thing even when Java will not do so automatically. This is accomplished by means of an ***explicit cast expression***. Like automatic coercion, a cast expression gives you a view of the Thing cast as a different type.

For example, if you have a Dog and know that it is actually a Collie, you can tell Java to treat it as a Collie. It would be an error to ask

```
myDog.herdSheep()
```

-- sheep herding is not an ability of most Dogs. But if you know that `myDog` is a Collie, you can tell Java to treat it as a Collie and then ask it to herd sheep. The syntax of this request -- which we will examine below -- is

```
((Collie> myDog).herdSheep()
```

or, more verbosely,

```
Collie aCollie;

aCollie = (Collie> myDog;

aCollie.herdSheep()
```

[[Footnote: The more succinct -- and preferred -- form of the Collie cast actually begins with two parentheses. One -- the pair around Collie -- is the cast expression. The other -- surrounding (Collie) myDog -- is used for grouping as described in the final section of this chapter.]]

When you use an explicit cast expression, you must be sure that the thing you're casting really can be coerced to its new type: for example, that the Dog you are casting really is a Collie. The explicit use of a cast expression tells Java that you really do mean to do something that Java can't tell is OK, like make a lossy coercion. It is your responsibility to be sure that the request you are making really is OK. An `instanceof` expression can be used to verify type membership before a cast expression.

The syntax of a cast expression is

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

That is, you put 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 dial displaying the value `3` -- perhaps from

```
int myInt = 3;
```

-- then

```
(long) myInt
```

is a `long`-sized dial displaying `3` 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 dial displaying the value `3`. Casting, like implicit coercion, does not actually modify the castee.

f explicit cast:
pe) oldThing
oldThing*; type is *NewType

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`.

Explicit cast expressions are also allowed from one Java object type to another under certain circumstances. Specifically, a Java object may be cast to a type if the object is an `instanceof` that type.

For further details on explicit cast expressions, see the sidebar on Coercion and Casting in Java.

5.5.2 Equality Testing and Identity

The final primitive- and object-Thing operation is equality testing. There are actually two equality operators that can be used with either primitive or object Things: `==`, which tests whether two Things are the same, and `!=`, which tests whether two Things are different.

Beware: `==` tests for equality; `=` is the assignment operator.

```
expression1 == expression2
```

is an expression with boolean type; its value is true if *expression2* are the same and false otherwise.

```
expression1 != expression2
```

also has boolean type, but its value is false if *expression2* are the same and true otherwise.

Java Notes

Java Operators

Java operators include

```
+    -    *    /    |    &    ^    %    <<   >>
+=   -=   *=   /=   |=   &=   ^=   %=   <<=  >>=
<    >    <=   >=   ==   !=
!    &&   ||
++   --
=    ?:
```

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

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

The third row above lists the six comparison operators, eac of which returns a boolean. The final comparison is not-equal.

The fourth row lists the logical operators: logical negation, logical conjunction (and), and logical disjunction (or). Each of thes takes boolean arguments -- one in the case of negation, two in the case of conjunction and disjunction -- and returns a boolean.

The operators in the fifth 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++` return the unincremented value. The `--` operator works similarly.

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

```
x > y ? a : b
```

evaluates to `a` if `x > y` and to `b` otherwise. (Any boolean-valued expression may be used instead of `x > y` and any expression may b used in place of `a` or `b`.)

`ion1 == expression2` Type is boolean; true iff *expression2* are the same. `expression1 !=` `ion2` Type is boolean; true iff *expression2* are different.

Wh are two Things the same?

- For primitive types, Things are the same whenever they "look" the same, i.e., when their (types are compatible and) values are indistinguishable. [[Footnote: Java actually performs any automatic coercions available to compare primitive objects, so `(int) 3 == (long) 3` is true.]]
- For object types, values are the same exactly when the two expressions refer to the *same object*.

This last point is actually rather subtle. Two objects may look different but actually be the same, or they may appear similar but actually be different. Consider, for example, identical twins *`x`* and *`y`*. Although they may look exactly the same, they are still two different people. If one gets a haircut, the other's hair doesn't automatically get shorter. If one takes a bath, the other doesn't get clean. Thus they are different: *`x`* `==` *`y`* is `false`.

For Java equality testing, 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. On the other hand, two object Things can be the same even if they happen to be viewed through different types: my CellTelephone and your Telephone might actually be the identical object.

Equality of primitive Things is different. The `int 3` has no internal structure that can be changed (the way that one twin's hair can be cut). If you change `3`, you don't have `3` any more. If dial name *`a`* and dial name *`b`* are each of type `int` and each has `3` as its value, then *`a`* `==` *`b`* is `true`.

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. Assignment and equality testing, described above, are also Java operators. Of special note are + for String concatenation and unary - for negation.

f binary operation:
`peration otherThing`
bar for type, value.

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 + y`. 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. The operands of this operator can be any expressions, not just primitives or objects.

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 Arithmetic Expressions, an expression of the form

erator type has type ***type***

```
type operator type
```

has type *`type`* for all of the basic arithmetic operations on most of the primitive types. That is, 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, the expressions

```
3 + 7

2.0 * 5.6

5 / 2
```

evaluate to the `int 10`, the `double 11.2`, and -- perhaps surprisingly -- the `int 2`, not `2.5` (or `2.0`), respectively.

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 dial 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 in Java.

Other arithmetic operators work in much the same way as `+`. Additional information on arithmetic

Java Notes

Types of Arithmetic Expressions

Arithmetic expressions include the binary operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and the modulus or remainder operation (`%`). In addition, there are two una 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`, Jav 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 in Java.

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 operator `+`, `-`, `*`, and `/` are the sum, difference, product, and quotient of their (possibly widened) operands, respectively.

The value of `x % y` is the (appropriately widened) remainde when `x` is divided by `y`.

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

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

```
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
```

and (best style)

```
7.0 / 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 a (literal) `int`, while `- 5` is an arithmetic expression that has value `-5` and type `int`. (Subtle, no?)

5.6.2 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 `==` and `!=`, which have already been described, as well as more specialized numeric comparators such as `<` and `<=`. The sidebar on Java Operators contains a complete list.

Each of these operations takes two numbers, coerces them appropriately, and then returns 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`.

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, denoted `!`. The Java fragment

```
!(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.)

Java Notes

Other Assignment Operators

Compound Assignment

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

```
total = 6
```

or

```
total = total + 1
```

(The second expression 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 assignme operators***. It works by computing the value of its first operand, the 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 cours There are also other compound assignment operators in Java, including `-=`, `*=`, `/=`, and `%=`. Like the `+` operator, 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).

5.6.3 AutoIncrement and AutoDecremen

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: `++var` first increments, then produces a value; `var++` produces the value first.) The two (prefix and postfix) autodecrement operators work similarly.

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). 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, suppose that the name `x` has the value `6` and consider the following expression:

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

The person who wrote this expression might well have meant

```
"I have " + (x + 3) + " monkeys"
```

which evaluates to `"I have 9 monkeys"`. However, Java evaluates the expression by grouping subexpressions from the left, more like

```
( ( "I have " + x ) + 3 ) + " monkeys"
```

This evaluates to `"I have 63 monkeys"`! Isolating `x + 3` with parentheses makes the `+` in `x + 3` behave as 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 matter. For example,

```
2 + 3 * 2
```

doesn't have the same value as `5 * 2` --

```
2 + 3 * 2
```

is `8`. 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`. (Java evaluates the * first.)

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.

Chapter Summary

- Evaluating an expression produces a Thing with a type and a value. Each kind of expression has its own rules of evaluation that determine the type and value of the Thing it produces.
- Simple expressions include literals and names.
  - A literal has its apparent type and value.
  - A name has its declared type and assigned value.
- Assignment expressions are generally used for their effects -- modifying the value associated with a (label or dial) 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 only on objects:
  - A method invocation expression asks an object to perform an action. It 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.
  - An instance creation expression's value is a brand new object whose type is the (class) type with which the constructor expression is invoked.
  - An type membership -- instanceof -- expression tells you whether an object can be treated as a member of a particular (class) type. Its type is `boolean`.
- Some expressions can operate on either primitive Things or objects:

    - An explicit cast expression's value is the same as the Thing it is given, but with a different type.
    - Equality testing expressions produce boolean values.
        - Two primitive Things are the same if they appear to be the same.
        - Two objects are the same only if they are actually the same object.
- 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 `boolean`.
    - Explicit cast expressions have the type of the cast operation and the same value as the cast operand.
    - None of these operations actually modifies any of its operands. However, autoincrement, autodecrement, and the shift operators *do* modify their operands.
- A compound expression contains multiple sub-expressions. It is evaluated by evaluating each of its constituent parts, then combining the resulting Things.
- Parentheses can be used to group expressions. Otherwise, expressions are evaluated in the order of precedence established by Java.

Exercises

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

```
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.

    a. `6`

b. `24L`

c. `+3.5`

d. `3.5f`

e. `2e-16`

f. `-25b`

g. `i`

h. `i+3`

i. `i+3.0`

j. `i+s`

k. `l+d`

l. `f+s`

m. `i / 0`

n. `4 * 3.2`

o. `i = 0`

p. `i == 0`

q. `b = 0`

r. `b == 0`

s. `'c'`

t. `"An expression in double-quotes"`

u. `"An expression in double-quotes" + "another one"`

v. `"6" + 3`

w. `!b`

x. `!i`

y. `b || true`

z. `i += s`

aa. `s += i`

ab. `i += f`

ac. `l = i = s`

ad. `i = l += s`

ae. `l++`

af. `(long) s`

ag. `s`

ah. `(short) l`

ai. `l`

2. Give examples of three expressions with side effects.

3. For this question, you may wish to consult the sidebar on Java Operator Precedence.

   Assume the following definitions:

```
int i = 93;

boolean b = true;
```

   What is the value of each of the following expressions? Which ones produce errors in evaluation?

   a. 2.0 + 3.5 * 7

   b. ("top " + "to " + "bottom").toUpperCase()

   c. "the answer is " + 6 * 7

   d. 4 + 6 + " is " + 10

   e. i > 0 && i < 100

   f. b = i < 0

   g. ! (i == 0) && 100 / i

4. Assume that `x` and `b` are previously defined names for an `int` and a `boolean`, respectively. Give examples of each of the following:

   a. An expression whose type is int and whose value is more than a previously defined x.

   b. An expression whose type is boolean and whose value is true when x is between 5 and 15.

   c. An expression whose type is double and whose value is half of x's.

   d. An expression whose type is long and whose value is the remainder when x is divided by 7.

   e. An expression whose type is boolean and whose value is the opposite of the boolean b.

   f. An expression whose type is boolean and whose value is true exactly when the int x is evenly divisible by 5.

   g. An expression whose type is String and whose value is read from the user's keyboard.