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

Introduction to Interactive Programming

By Lynn Andrea Stein

A Rethinking CS101 Project

Chapter 3.
Things, Types, and Names

Chapter Overview

- What kinds of Things can computers talk about?
- How do I figure out what they can do (or how they interact)?
- How can I keep track of Things I know about?

This chapter introduces some of the conceptual structure necessary to understand Java programs. It begins by considering what kinds of things a program can manipulate. Some things are very simple--like numbers--and others are much more complex--like radio buttons. Many complex things can actually act, either by themselves (e.g. a clock that ticks off each second) or when you ask them to (e.g. a radio that can play a song on request). These complex things are called objects.

To manipulate objects, you need to understand two important concepts: types and names. A type tells you what kind of thing an object is. It tells you what kind of behavior you can expect from that object. A type is a way of setting expectations -- for the computer program and for the software developer -- about the object. In Java, the type of a thing lets the computer keep track of what that thing can do.

A name is a way of referring to a thing that already exists. A name doesn't bring a thing into existence, but it is a useful way to get hold of a thing you've seen before. Every name has an associated type, which tells you what sorts of things the name can refer to. It also tells you what you can expect of the thing that that name refers to.

In other words, a type describes how you can interact with the thing that a name names. There are actually two different kinds of names in Java: primitive (dial) names and reference (label) names.

Sidebars in this chapter cover the details of legal Java names, Java primitive types, and other syntactic and language-reference details.

Objectives of the Chapter

1. To recognize Java types.
2. To understand that documentation for a type tells you what objects of that type can do.
3. To be able to declare and define names.
4. To understand that a declaration permanently associates a type with a name.
5. To understand how a label name can have a referent or have no referent (i.e., be null).
6. To be able to tell when the values associated with two names are equal.
7. To distinguish Java object from primitive types.
8. To recognize that each dial name contains exactly one value at any time.

3.1 Things in Programs

In the first section of this book, we have explored what computational systems are and how they are created. In this section, we shift our focus to the pieces out of which such systems are created. The previous section took a top-down approach to system design. In this section, we learn the basic building blocks that enable us to also approach the problem from the bottom up.

In building a computer program, we will make use of many Things. Some of these Things are relatively simple and, when we need them, we can just write them down. For example, `4` and `-6.3` are Things we might want in our program, and we can put them there just by writing them down in the appropriate places. But other Things -- like the library's record for *Moby Dick* or the current location of the cursor on the computer screen -- are harder to write down explicitly in the way that we are able to write `4` or `-6.3`. For these kinds of Things, we need a way to refer to them without explicitly writing them down each time. We do this using a name.

A ***name*** is a way to refer to something in your program. At any given time, a name refers to at most one Thing. What the name refers to may change over time, though, so the current location of the cursor may

be at coordinates (26, 155) now and at (101, 32) later. This is true in real life, too: *the student whose birthday comes next* changes every time someone celebrates becoming a year older.

In this chapter, we explore how names refer to Things in Java. We also explore some specific kinds of Things that can be included in a Java program. Although there are details in this chapter that are particular to Java, many of the principles we identify work in almost any computer programming language. These include:

- Names can be used to refer to Things, allowing you to hold on to a Thing even if you can't spell it out entirely.
- Some Things can be written directly into your programs. These are called literals.

In addition, in Java and in many other languages, each name can be used only to refer to Things of a particular type. A language in which a name is restricted to refer to Things of a particular type is called a ***strongly typed language***. In a strongly typed language, like Java,

- A name must be declared, meaning that you must say explicitly in the program what type of Thing this name can refer to.

We will learn more about types and names below. To begin, let's look at Things themselves. What kinds of Things exist in computer programs?

[[Footnote: You may have noticed that the word Thing is capitalized almost every time that it appears in this chapter. This is to indicate that, in this chapter, we are using it in a very particular way. In Java, no one word refers to all of the objects -- like KlingonStarships and BookIDs -- and also all of the simpler things in programs -- primitive data like 6 and -43.5 -- at the same time. To avoid using another word incorrectly or imprecisely, we'll use Thing with a capital "T" to refer to all of the entities, objects, and primitive data that appear in programs.]]

3.1.1 Most Java Things are Objects

We have already seen a number of different kinds of program Things. Some -- like `4` and `-6.3` -- probably look quite familiar. Others, like *Moby Dick* or the current location of the cursor on the computer screen, may look different once we represent them inside the computer, but they, too, will be Things we can talk about in our Java programs.

Different programming languages allow you to talk more or less easily about different kinds of Things. Some languages -- like Basic -- are very restricted in the kinds of Things that a program can talk about explicitly. Other languages -- like Java -- allow you to create and talk about almost any kind of Thing that you can imagine. Java is one of a family of languages called ***object oriented programming languages***. In Java, as in most object oriented programming languages, most Things -- and all of the Things that you will create -- are called objects. Almost any Thing that you can describe in a programming language can be an object, so, in Java, ***object*** is a pretty general word for "programming language Thing". Not all programming languages allow you

to talk about objects, though, and -- towards the end of this chapter -- we will learn about a few Java Things that are not objects.

An object can be just about anything that you might want to represent in a computer program. Some example objects include *the radio button that the user just clicked*, *the window in which your program is displaying its output*, or *the url of your home page*. The Klingon starship racing across your computer screen -- in a Star Trek game -- is a Thing that the computer has to keep track of. If your Star Trek game is written in Java, that Klingon starship is almost certainly an object.

Objects are frequently complex Things with internal state. For example, the window may have a "close me" control or a background color; the url has a host computer name. Many objects also know how to act and can perform tasks. The radio button may be able to tell you whether it has been selected, or the window may know how to close. The library checkout desk of the previous chapter keeps track of book circulation while the card catalog can tell you who wrote what books. And, of course, the Klingon starship has a position and a velocity and a certain amount of ammunition left to shoot at you with....

Some objects can even act on their own without being asked to do anything; they are "born" or created with the ability to act autonomously. For example, an `Animator` may paint a series of pictures rapidly on a screen, so that it looks to a human observer like the picture is actually moving. The `Animator` may do this independently, without being asked to change the picture every 1/30th of a second. Similarly, an alarm clock may keep track of the time and start ringing when a preset time arises. A starship may move -- changing its position on the screen -- or shoot at you or even explode.

Objects are useful because we can ask them to act -- or have them take actions on their own. But how do we know what we can do with a particular Java object? In order to use an object, we need to know its type.

3.1.2 A Type Tells You What You Can Do With A Thing

A type tells you what kind of behavior you can expect from an object. We have been using the idea of types all along, but we have not yet made it concrete and specific. For example, when we talk informally about a library book's representation in the computer or even something simple, like a number, we mean to be referring to a specific kind of Thing. In a programming language, we also talk about categories of Things this way. In English, we can refer informally to the kind of Thing, like "a number". In a programming language, we need to say what we mean by a number. In this chapter, we'll be using types that someone else has defined. In the next several chapters, we'll be learning about how to figure out what a certain type of Thing is like. In chapter [Classes and Objects], we will at last learn how to build our own types of Things.

3.1.2.1 What a Type Is

A type specifies what a Thing can do (or what you can do with a Thing). Types are like contracts that tell you what kinds of interactions you can have with Things. What an object's structure is -- and what that object can do, or what you can do with it -- is fully determined by its type, i.e., what kind of object it is.

In the previous chapter, we saw a number of Things that make up a library system: the circulation desk and the card catalog, book IDs and patron IDs. If we were to write Java code corresponding to the system that we designed there, we would make each of these kinds of Thing a type of object. The description for the type CirculationDesk would tell us that a CirculationDesk can checkIn a bookID or checkOut a bookID to a patronID, for example. In other words, the type of the Thing -- CirculationDesk -- tells us what a particular CirculationDesk -- like the Smalltown Library's -- can do.

Java itself has certain predefined object types, such as `DialogBox` -- the little window that pops up on your computer screen to give you an urgent system message -- and `URL`-- the internet address of a web page. If you are using the cs101 course libraries, you'll also have access to object types such as `AnimateObject`-- something that can move by itself -- and `DefaultFrame`-- an easy kind of user interface window to use.

Other object types may include `KlingonStarship` (if you're building a space battle adventure game), `IllustratedBook` (if you're building an electronic library system), or `PigLatinTranslator` (if you're building a networked chat program). In the rest of this book, you will be learning to define object types -- and create instances of those types -- to do what you want. These types -- whether a part of the Java language, included in a library you choose to use, or of your own defining -- are all kinds of objects. Note that, by convention, the name of each object type -- each class -- starts with a capital letter.

3.1.2.2 Types and Instances

A type itself doesn't do anything; it is *objects of that type* that are generally useful in one's programs. These individual objects are sometimes called ***instances*** (of the type). So `KlingonStarship` is a type, but the particular Thing streaking across your computer screen, phasers blasting, is an individual `KlingonStarship` instance.

A particular object types may describe many different individual objects -- the three specific `KlingonStarships` visible on your screen, the five hundred and seven `IllustratedBooks` in the children's library, or the particular `PigLatinTranslator`that your particular chat program is using. These instances share a type -- they are the same *kind* of Thing -- but are independent objects. For example, the `KlingonStarship` that you just destroyed is a different `KlingonStarship` instance from the one that is getting ready to fire its phasers at you. We will explore this idea in greater detail in chapter 7. By convention, an individual object -- as opposed to its type -- is generally given a name that starts with a lower case letter.

Each kind of object -- each object type -- determines what individual objects of that type can do. For example, windows can close; dictionaries can do lookups; KlingonStarships can fly around the screen. Each particular kind of object provides a particular set of services or actions that objects of that kind can do. Further, each individual object of that type can perform these actions. For example, if `myWindow` and `yourWindow`are two different window-type objects, `myWindow` can close, and so can `yourWindow`. But if `myWindow` closes, that doesn't in general affect `yourWindow`.

3.1.2.3 Asking for Action

Each individual object comes ready-made with certain type-specific properties and behavior.[[Footnote: It is tempting to say that this is *typical* behavior, and indeed this is just what the word typical means: pertaining to the type.]] An `IllustratedBook` has an author and an illustrator, for example. A `PigLatinTranslator` may be able to translate a word that we supply it into Pig Latin. We ask objects to act (including telling us about themselves) using specific services -- behaviors -- that these objects provide.

A common thing to do in a computer program is to ask a particular object to perform a particular service.[[Footnote: Also to engage in a specific behavior, to take a particular action. As we shall see, the formal name for this is "method invocation".]] An object's services are requested by giving the name of the object we're asking followed by a dot (or period), followed by the request we're making of the object. So if `theLittlePrince` is the name of an `IllustratedBook`,

```
theLittlePrince.getAuthor()
```

would be a request for the name of the author of the book: "Maurice de Saint Exupery". Similarly, if `myTranslator`is a `PigLatinTranslator`, `myTranslator.processString("Hello")` might be a request to `myTranslator` to produce the Pig-Latin-ified version of "Hello", which is "ello-Hay". We will explore this service-requesting further in the next few chapters, but for now you can regard it as a special incantation to use when you want to ask a Thing for something that it knows how to do for you. These requests are the most basic form of interaction among the entities in our community.

Sometimes, the same Thing can be viewed in different ways, i.e., as having multiple types. For example, a person can be viewed as a police officer or as a mother, depending on the context. (When making an arrest, she is acting as a police officer; when you ask her for a second helping of dessert, you are treating her as a mother.) A Thing's type describes the way in which you are regarding that Thing. It does not necessarily give the complete picture of the Thing.

3.1.3 Some Useful Types of Things...

Now that we have seen how objects and types work, let's look at some useful objects and object types. We have begun -- this chapter and your programming experience -- by using objects and types that are provided to us, because it is important to learn how to interact with an object. Many of the objects that you will use throughout your programming career will be designed by other people for you to use. Later -- in chapter [Classes and

Java Notes

Selected String Behavior

Below are some selected services provided by individual Strings, along with brief explanations and examples of their usage.

`toUpperCase()` produces a String just like the String you start with, but in which all letters are capitalized. For example, `"MixedCaseString".toUpperCase()` produces `MIXEDCASESTRING`

`toLowerCase()` produces a similar String in which all letters are i lower case. So `"MixedCaseString".toLowerCase()` produces `mixedcasestring`

`trim()` produces a similar String in which all leading and trailing white space (spaces, tabs, etc.) has been removed. So

Objects] -- we will learn how to build objects of our own.

One particularly useful kind of object -- built in to Java and to most programming languages -- is called a `String`. A ***string*** is a piece of text, a sequence of characters. To describe a specific string -- for example, the message that your computer prints to its screen when you first boot it up -- you can write it out surrounded by double quotation marks: `"Hi, how are you?"` or `"#^$%&&*%^$"` or even `"2 + 2"` are strings. Note that the quotation marks are not actually part of the string; they're just there to make it clear where the string begins and ends. Your computer doesn't *understand* the string, it just remembers it. (For example, the computer doesn't know of any particular relationship between the last example and the number `4` -- or the string `"4"`.)

Strings are useful, for example, to communicate with a program's user. Error messages, user input (i.e., what you type to a running Java program), titles and captions are all examples of strings. Strings can perform some (moderately) interesting tasks. For example, if `myName` is "Rigoberto Manchu", then `myName.toLowerCase()` is "rigoberto manchu" and `myName.length()` is 16. Strings are part of the Java language, so they are available to every Java program. For more information on the `String` type and what it can do, see the sidebar on Selected String Behavior.

Another useful object is `Console`, described in the next sidebar. `Console` is an object that can print a String to the ***Java console***, a standard place on the computer screen where someone running a Java program can look for information. `Console` can also readln a String that the user types to the Java console.

`"  a very spacey String  ".trim()` is just `"a very spacey string"`

`substring( fromIndex )` produces a shorter String containing the same characters that you started with, but beginning at index *`fromIndex`*. Bear in mind that the index of the first character of a String is 0. `substring( fromIndex, toIndex )` produces the substring that begins at index *`fromIndex`* and ends at `toIndex - 1`.

`"Hello".substring(3)` is `"lo"`

`"Hello".substring(1,4)` is `"ell"`, and

`"Hello".substring(0)` is `"Hello"` again.

`length()` returns the number of characters in the String. For example, `"Tee hee!".length()` is 8. Since the String is indexed starting at 0, the index of the final character in the String is the String's `length() - 1`.

`replace( old, new )` requires two characters, *`old`* and *`new`*, and produces a new String in which each occurrence of *`old`* is replaced by *`new`*: [[Footnote: A character is, roughly, a single alphanumeric or symbolic character (one keystroke) inside single quotation marks. For more detail on what exactly constitutes a character, see the chapter on Java types.]] For example,

`"Tee hee!".replace('e', '*' )`

produces

`"T** h**!"`

`charAt( pos )` requires an index into the String and returns the character at that index. Recall that Strings are indexed starting at 0.

`"Hello".charAt( 2 )` is the same as `"Hello".charAt( 3`

`indexOf( character )` returns the lowest number that is an index of *`character`* in the String.

`"Hello".indexOf( 'H' )` is 0 and

`"Hello".indexOf( 'l' )` is 2. Also,

`"Hello".indexOf( 'x' )` is -1, indicating that 'x' does not appear in "Hello".

`lastIndexOf( character )` returns the highest number that is an index of *`character`* in the String.

`"Hello".lastIndexOf( 'H' )` is 0 and

`"Hello".lastIndexOf( 'x' )` is -1, but

`"Hello".lastIndexOf( 'l' )` is 3.

Strings are unusual among object in being able to be written out this way. Most objects can't be spelled out like this; they need to be explicitly created in a somewhat different fashion. [[Footnote: We will see how to do this in the chapter on Classes and Objects]] Strings are also unusual in a second way: once created, a String cannot change. If you use one of the String services described here that provides a String as a result, that result is a *new* String differing from the original. (The original remains, unchanged.)

Unlike `String`, `Console` is not built in to Java. It is a part of the cs101 libraries. This means that `Console` is only available to Java programs that use the cs101 libraries, a group of free code libraries developed for use with this textbook. Later in this book, we'll learn how to replicate the function of `Console` using only Things built in to Java, in case you don't always want to use the cs101 libraries.

There is another difference between `Console` and `String`: `String` is a *kind* of object -- a type -- while `Console` is actually a particular object. `Console` has a type, of course, but you will not be interacting with other similar objects. That is why the Console sidebar gives you information about the particular object, `Console`, but the String sidebar talks generically about all `Strings`.

Once you know the name of a particular object -- like `Console` -- and you know about the type of Thing that object is, you can ask it to perform tasks for you. For example, you can ask `Console` to print out "Hi there" on the screen. In Java, this is spelled `Console.println("Hi there")`. (Recall that the . and () are special syntax -- notation -- that indicate that we're asking the Thing called `Console` to do its `println` behavior (on the String `"Hi there"`).) Specific information about the Things that you can do with `Console` is contained in the sidebar. In the next chapter, we'll explore in greater depth how you can find out what kinds of actions any particular kind of Thing can perform.

3.2 Doing Things With Things

A few objects, like certain `Strings`, can be typed directly on your keyboard. If you want to use a particular `String` in your program, one option is to type it in right there, as we have done in the sidebar. Other objects, like `Console`, are always available for use through their names. When you want to use a particular object, you will need to figure out how to identify it so that you can use it in your program. A name is one way to do this, but it is not the only way.

A name -- like `Console` -- gives you direct access to the object it names. For example, I might have a robot that I call `Robbie`; then I can ask `Robbie` to move using its name. I can also tell you about `Robbie` using this name. Below, you will see how to give names to Things, and -- using that technique -- I could even give `Robbie` the nickname `Fred` by giving the robot a second name. A name like

Java Notes

Console

Console is a special object that knows how to communicate with the user in some very basic ways. If your program says

```
Console.println( "Hello there!" );
```

Then the String "Hello there!" will appear on the Java console. (Remember, the double quotes aren't part of the string; they're just used to indicate where it starts and ends.) The statement

```
Console.print( "Hi" );
```

is similar, except that `Console.print` doesn't end the line of outpu while`Console.println` does. This means that

```
Console.print( "A " );
Console.print( "is for apple." );
```

would produce the output

```
A is for apple.
```

while

```
Console.println( "A " );
Console.println( "is for apple." );
```

would produce

```
A
is for apple.
```

`Robbie` or `Console` names the object, and is like a label that allows you to access the object directly.

Imagine that, in your program, `mobyDick` names a book. You might be able to refer to the author of Moby Dick by using a name designated for this purpose, like `hermanMelville`. You can also refer to an object without using a name for it. You can do this, for example, by using a related object to help you. So, for example, `mobyDick.author()` might refer to the author of that book. (As with `Console.println("Hi there")`, above, the . and () are special syntax. In this case, we are asking the Thing called `mobyDick` to tell us its author.)

You can of course combine `print`s and `println`s arbitrarily. Printing a String containing a special character called a newline character escape (spelled `\n`) causes the line to end as well.

You can also use Strings that are associated with names or any other Strings you may have access to, not just String literals.

`Console` provides one other important service, `Console.readln()` (pronounced "read line") which takes no arguments and returns a `String`, specifically the `String` typed by the user (and ending with a return or enter character) on the Java console. [[Footnote: To use Console, you must install the cs101 libraries in your Java system and import the cs101.util.Console package. Your instructor may have set this up for you or can show you how to do so yourself. In all of the above examples, you could replace Console by the name System.out, which is part of the standard Java libraries (and hence needs no special action on your part to use it). However, Console also provides basic input facilities - such as readln() -- not present in System.out.]]

In this way, we can access an object -- like `hermanMelville` -- indirectly -- through `mobyDick`. This is useful if we don't have otherwise have a name for it. (The same trick works in English, too: *Omoo* was written by *Moby Dick*'s author.)

There are differences between direct and indirect reference. One is the length and complexity of the way we say it: `HermanMelville` vs. `mobyDick.author()`. Another has to do with whether the referent -- the object referred to by the name or the service request -- is expected to change. Names like `hermanMelville` (or `Console`) might always refer to the same Thing, as would `mobyDick.author()`. But some indirect references are expected to produce different Things each time the service request is made. Imagine that `fortuneTeller` is the name of an object that knows how to produce a random pithy saying (a fortune). The name `fortuneTeller` might even always refer to the same oracle. But you'd hope that a request like `fortuneTeller.getFortune()` would produce different pity sayings from one request to another, or the Thing named `fortuneTeller` wouldn't be very interesting!

3.3 Use Names to Keep Track of Things

With all of these Things floating around in our program, it is pretty easy to see that we'll need some ways to keep track of them. The simplest way to keep track of Things is to give them names. This is called ***assigning*** a value to a name. Giving something a name is sort-of like sticking a label on the Thing. We sometimes say that the name is ***bound*** to that value.

Unfortunately, not every Thing in our program comes with a name. Consider the fortune teller of the preceding section. We can ask the `fortuneTeller` to produce a fortune, but if we don't do anything with that fortune, we will have no way to refer back to it. It is as if, as soon as we get the fortune from `fortuneTeller`,

we drop it on the floor with all of the other Things that our program has created or used. Unless we somehow keep hold of what `fortuneTeller` gives us, we cannot get it back later.

We can solve this problem by putting a label on the first fortune when we get it. Then, if we want it back later, we can ask for it by name: the name on the label. In fact, that's just what names are for Java objects: labels that let us keep track of Things.

To actually assign a value to a name -- to create a binding between that name and that value, to stick the label on the Thing -- Java uses the syntax -- that is, the notation --

```
name = value
```

So we can remember the fortune our `fortuneTeller` provided by giving it a name:

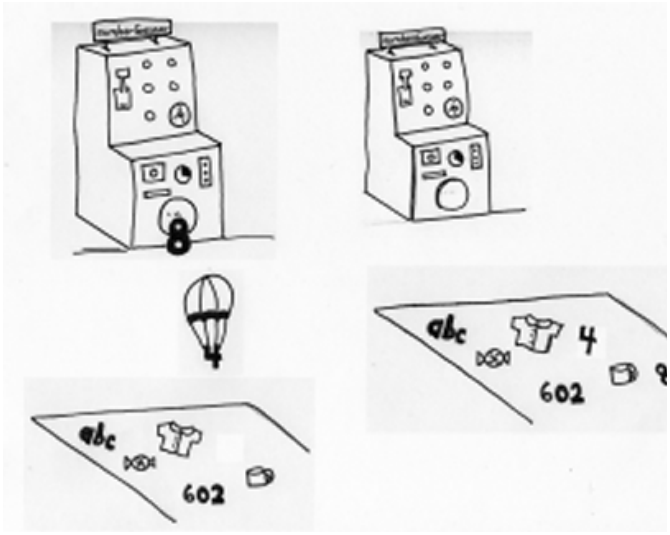


Figure 3.1. If we don't do something with our Things, they get mixed up with the other Things we used.

```
myPersonalFortune = fortuneTeller.getFortune()
```

This associates the fortune produced with the name `myPersonalFortune`. Once a particular name refers to a particular Thing, we can use the name wherever we would use its value, with the same effect:

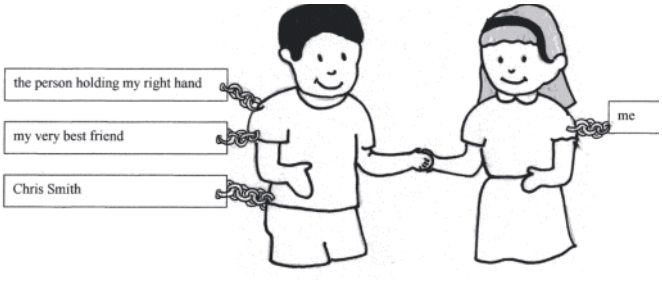
```
Console.print( "And your fortune is...." );

Console.println( myPersonalFortune );
```

The name becomes a stand-in for the Thing it refers to: the fortune told by the `fortuneTeller`. In the next chapter, we will see that a name is a simple kind of expression.

A name, like a label, can only be affixed to one Thing at a time. In other words, only one value may be associated with a name at any given time. One Thing can be referred to by any number of names at once (including, potentially, no names at all). The same person can be "the person holding my right hand", "my very best friend", and "Chris Smith". But only one person is "the person holding my right hand". [[Footnote: Barring weird interpersonal pileups, of course.]]

3.3.1 Declarations: Creating a Typed Name



But where do these names come from? Except for the rare name, like `Console`, that is available when your program is started -- and that is

probably already being used to keep track of something important -- you will need to come up with your own names and to tell Java about them, i.e., to generate the labels that you'll want to stick on Things. The way that you do this is with a declaration

Figure 3.2. A name refers to only one Thing at a time, but severa different names can refer to the same Thing.

We already know that each Java Thing comes into the world with a type, i.e., an indication of what kind of Thing it is. In Java and other strongly typed languages, every name -- and every label -- also has a type. A Java name can only be used to label objects of the appropriate type. This type is associated with the name when the name is created. **The type associated with a particular name never changes.**

Since every Java name must have an associated type, the way that you create a name in Java is to tell Java what that type is. The term for telling a programming language what type a name has is **declaration**. Declaring a name means stating that that particular name is to be used for labeling values (Things, objects) of some particular type. A declaration creates a name -- a label -- suitable for sticking on objects of that particular type. The label can only be stuck on things of that type.

Figure 3.3. A label is declared to have a particular type, meaning it is suitable for labelling things of that type.

Names are declared using the ***type-of-thing name-of-thing*** rule:

```
String bookTitle;

KlingonStarship enemyFighter;
```

The second word on each line is a name that is being declared. The first word on each line is the type that the name is being declared to have. In the first line of the example above, `bookTitle` is being declared to have type **`String`**. This is the Java type for a piece of text, like `"The Forgotten Beasts of Eld"`. Finally, each declaration ends with a semicolon (`;`). So the first declaration here creates a name, `bookTitle`, suitable for naming pieces of text (or, in Java, `Strings`). The second line creates the name `enemyFighter` and says that it can refer to any Thing of type `KlingonStarship`.

In both of these cases, a new name -- a new label -- is created. That is, each of `bookTitle` and `enemyFighter` is now a label of the corresponding type. But neither of these statements says what the label is stuck on. In fact, it's quite possible that the label is stuck on nothing at all. Later in this chapter, we will see a special kind of name that doesn't create a label. But in Java, objects are named with label names.

A name has a certain lifetime, sometimes called its ***scope***. Within that scope -- over its lifetime -- the name may be bound to many different values, though it can only be bound to one value at a time.

Java Notes

Java Naming Syntax and Conventions

Java identifiers can contain any alphanumeric characters as well as the symbols $ and _. The first character in a Java identifier

For example, `enemyFighter` may initially be the big ship right in front of you, but later change to be the high-speed ship just entering your field of vision. The association between a name and a type persists for the lifetime of the name, however. `enemyFighter` can only name a `KlingonStarship`, not a `String` or a `BookID`.

3.3.2 Definition = Declaration + Assignment

Declaring a name begins its useful lifetime, or scope. At that time, nothing else necessarily needs to happen -- and frequently, it doesn't. But sometimes it is useful to associate the name with a value -- to stick the label onto something -- at the same time that it is declared. This combination of a declaration and an assignment is called a ***definition***.

- A declaration creates a name, telling you what type is associated with a name.
- An assignment sets the value of a name, telling you what value that name is bound to.
- A definition combines the "what kind of Thing it can name" and "what value it has" statement types.

For example:

```
String bookTitle = "Weaving the Web";

String answer = Console.readln();

Person melville = mobyDick.author();

Cat myPet = marigold;
```

The first of these definitions makes use of a `String` literal, i.e., something that can just be typed in. It creates a label, `bookTitle`, and then immediately sticks it onto the `String "Weaving the Web"`. The second definition creates a label, `answer`, suitable for labelling `String`s, and sticks it on whatever the user types to the `Console`. The third definition uses `mobyDick`'s ability to tell us its own author. The final definition makes the name `myPet` refer to the same `Cat` currently named by `marigold`. This is a case of `marigold` standing in for the actual `Cat`, that is, the name being used in place of the Thing it refers to. After the assignment completes,

cannot be a number. So `luckyDuck` is a legitimate Java identifier, as is `_Alice_In_Wonderland_`, but `24T` is not.

Certain names in Java are ***reserved words***. This means that they have special meanings and cannot be used as names -- i.e., to refer to Things, other than any built-in meaning they may have -- in Java. Reserved words are sometimes also called ***keywords***. These are:

```
abstract boolean break byte case catch char class
const continue default do double else extends final
finally float for goto if implements import
instanceof int interface long native new package
private protected public return short static super
switch synchronized this throw throws transient try
void volatile while
```

Java is ***case-sensitive***. This means that `double` and `Double` are two different words in Java. However, you can insert any amount of ***white space*** -- spaces, tabs, line breaks, etc. -- between two separate pieces of Java -- or leave no space at all, provided that you don't run words together. You can't stick white space into the middle of a piece of Java -- a name or number, for example -- though.

Punctuation matters in Java. Pay careful attention to its use. Note, however, that white space -- spaces, tabs, line breaks, etc. -- does *not* matter in Java. Use white space to make your code more legible and easier to understand. You will discover that there are certain conventions to the use of white space -- such as lining up the names in a column, as we did above -- although these tend to vary from one programmer to the next.

`myPet` is bound to the actual `Cat`, not to the name `marigold`. But to understand this fully, we need to really think about what it means for a name to be a label.

3.3.3 Names are Labels

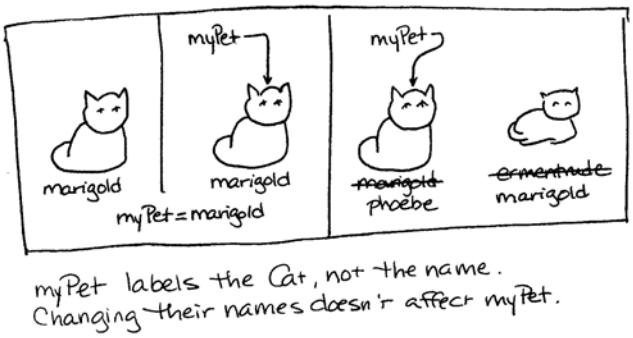


Figure 3.4. Marigold.

A Java object name really is just a label that that can be stuck onto an (appropriately typed) object. When a label-type name is declared, a new label suitable for affixing on Things with that type is created. For example, a building name might be a cornerstone label, a person's name might go on a badge, and a dog's name might belong on a collar. You can't label a person with a cornerstone or pin a badge on a dog, at least not without raising an error. Unlike cornerstones or dog tags, though, labeling a Java object doesn't actually change that object. It just gives you a convenient way to identify (or grab hold of) the object.

In Java terms, if we declare

```
RadioButton myButton;
```

this creates a label, `myButton`, that **can be** stuck onto Things of type `RadioButton`. Note that is **not** currently so stuck, though. At the moment,`myButton` is a label that isn't stuck to anything. Cornerstones and badges and dog tags don't come with buildings and people and dogs attached, either. Having a label is different from having something to label with it. **Labels don't (necessarily) come into the world attached to anything.** The value of a label not currently stuck onto anything is the special non-value ***`null`***. That is,`null` doesn't point, or refer, to anything. So the declaration above is (in most cases) the same as defining

```
RadioButton myButton = null;
```

Of course, we can attach a label to something, though we need to have that something first. We'll return to the question of where Things come from in a few chapters. For the moment, let's suppose that we have a particular object with type `RadioButton`, and we stick the `myButton` label onto it. (Now `myButton`'s value is no longer `null`.)

The value of a not-yet-stuck label is null.

After we give `myButton` a value -- stick it onto a particular `RadioButton`-- we can ask it whether it is currently pressed:

```
myButton.isSelected()
```

Figure 3.5. A label name that's not yet stuck on anything.

This request will return either `true` -- yes, I'm pressed -- or `false` -- no, I'm not. Let's imagine `myButton` currently is pressed, so it returns `true`. [[Footnote: This is an expression that returns a boolean value; see the discussion of expressions in chapter [Expressions].]]

Now, imagine that we declare

```
RadioButton yourButton = myButton;
```

The result of this declaration is that a new label is created. This new label is attached to the *same* object currently labeled by `myButton`. **Assignments of label-type names do not create new (copies of) objects.** In this case, we have two labels stuck onto exactly the same object, and we say that the names `myButton`and `yourButton` ***share a reference***. This just like saying that "the morning star" and "the evening star" both refer to the same heavenly body.

Because `myButton` and `yourButton` are two names of the same object, we know that

```
myButton.isSelected()
```

and

```
yourButton.isSelected()
```

will be the same: either the button that both names label is pressed, or it isn't. But we can separate the two labels -- say

```
myButton = someOtherButton
```

-- and now the values of

```
myButton.isSelected()
```

and

```
yourButton.isSelected()
```

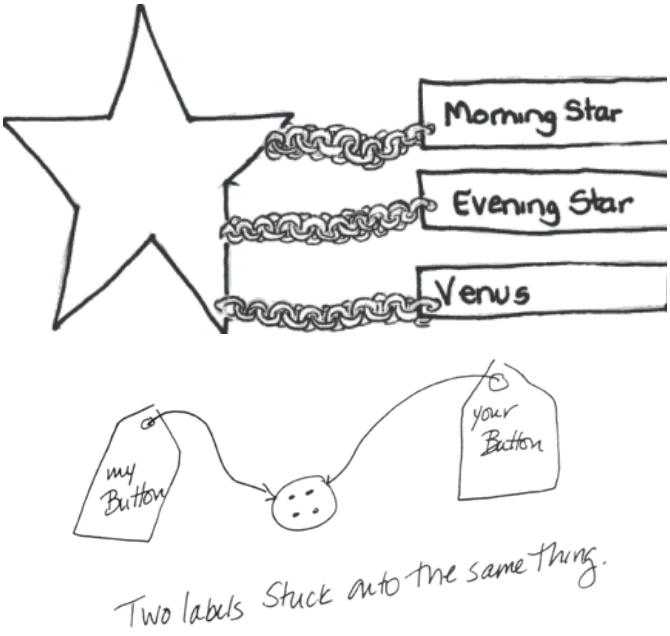


Figure 3.6. Multiple labels can refer to the same object.

might differ (unless, of course, `someOtherButton` referred to the same Thing as `yourButton`). Note that moving the `myButton` label to a new object doesn't have any effect on the yourButton label.

Note also that the labeled object is not in any way aware of the label. The actual radioButton doesn't know whether it has one label attached to it, or many, or none. A label provides access to the object it is labeling, but not the other way around.

3.4 A Tale of Things and Names

Let's walk through an example of how Things are named and how names refer to Things. This example involves a story that took place some years ago at a fancy party, the kind of party where one leaves one's hat at the door and whose attendees might include a few famous names....This story concerns one such party and three such personages: Charlie Chaplin, King George VI of England, and Eleanor Roosevelt.

First among our characters to arrive was Charlie Chaplin. He was wearing his usual bowler hat.

```
Hat charlieChaplinHat;
```

The code above just tells us that `charlieChaplinHat` is a label suitable for naming a `Hat`, but we will also assume that the the `charlieChaplinHat` label is, as usual, stuck on the bowler:

When Chaplin arrived in the lobby, he saw the hat check. He took off his hat and handed it to the hat check. (Fortunately, the hat check had a `checkHat` service, requiring a `Hat`:

Figure 3.7. We'll assume that the label `charlieChaplinHat` starts out stuck on Chaplin's famous bowler.

```
hatCheck.checkHat( charlieChaplinHat );
```

Of course, now Charlie Chaplin wasn't wearing a hat:

```
charlieChaplinHat = null;
```

So unburdened, he walked in to the party.

Next to arrive was the King of England. He was wearing his crown.

```
Hat kingGeorgeHat;
```

Again, we assume that -- prior to the execution of the rest of this code -- `kingGeorgeHat` is labelling the crown.

When King George arrived at the hat check, he, too, removed his hat and gave it to the hat check.

```
hatCheck.checkHat( kingGeorgeHat );

kingGeorgeHat = null;
```

[[Footnote:

Q. What would have happened if the King had executed these lines in the opposite order:

```
kingGeorgeHat = null;
```

```
                hatCheck.checkHat( kingGeorgeHat );
  ?
```

]]

Then he, too, went in to the party. Both men had a lovely evening at the party. Mr. Chaplin left first. Reentering the lobby, he approached the hat check and observed that it also provided a `returnHat()` service. So he executed

```
charlieChaplinHat = hatCheck.returnHat();
```

Much to his surprise, the hat handed to him by the hat check was not his simple black bowler. Instead, it was the magnificent crown of the King of England. Chaplin was still staggering around the lobby under the weight of this crown when King George emerged from the party and approached the hat check.

```
kingGeorgeHat = hatCheck.returnHat();
```

I'm sure it won't surprise you to hear that the King now found himself in possession of a simple black bowler. Being a man of simple good taste and few pretensions, George was half tempted to leave with that hat, but it occurred to him that there might be some other gentleman who would then regret the loss of such a sturdy topper. Further, he knew well from experience that the Crown of England can be a heavy burden and he hated the thought that someone else might have to suffer under it. Turning, he saw Charlie Chaplin staggering under just that burden.

"Sir," King George observed, "It seems our headgear has been exchanged by the hat check. Perhaps we should remedy this situation."

And so the two men proceeded to try just that. But there was a problem. Between them, the men had only two Hat labels, and each was occupied. For example, they considered executing

```
* kingGeorgeHat = charlieChaplinHat;
```

but this would have resulted in both men holding the crown, and Mr. Chaplin's trusty bowler lost in limbo with no way to retrieve it (since it would have been labelless).

Q. Explain why this would have been a bad idea.

The two men were puzzling over this dilemma when who should emerge from the party but Eleanor Roosevelt. (She'd snuck in while the reader wasn't looking.) This savvy diplomat immediately took measure of the situation

"I see that you two gentlemen have run into a bit of difficulty. Perhaps I can be of some assistance. You see, I am not wearing a hat, and so my head can be a temporary resting place while the two of you work your exchange.

"First, we will need a label for my hat:

```
Hat eleanorRooseveltHat;
```

"You see, of course, that there's no `Hat` there: `eleanorRooseveltHat` is `null`. It's simply a label that *could* be stuck on a `Hat`; it's the potential `Hat`-holder representing my head."

[[Footnote: Actually, what Mrs. Roosevelt really said was, "As you can see, `eleanorRooseveltHat == null` is now true, but of course you won't be introduced to ==, the identity operator, until the next chapter."]]

Next, the erstwhile Mrs. Roosevelt offered to take the Crown of England from Mr. Chaplin, to which he readily agreed:

```
eleanorRooseveltHat = charlieChaplinHat;
```

Now, the tall thin diplomat and the short comic actor found themselves jointly holding England's crown. At this, Mrs. Roosevelt suggested that Mr. Chaplin release the crown by seizing instead the simple black bowler to which he was accustomed (and of which the King of England was growing overly fond).

```
charlieChaplinHat = kingGeorgeHat;
```

Mrs. Roosevelt now had sole possession of the crown; Mr. Chaplin and King George both held the bowler. At this, King George released the bowler to take possession of his crown:

```
kingGeorgeHat = eleanorRooseveltHat;
```

Finally, Mrs. Roosevelt released her hold on the crown, freeing her to return to important business:

```
eleanorRooseveltHat = null;
```

(Of course, this step was not strictly necessary as each gentlemen now held the proper hat. Mrs. Roosevelt simply wanted to leave open her options for wearing other hats later.)

And the three figures left the party satisfied that all was well.

3.5 Primitive Types, Literals, and Dial Names

Objects are extremely useful and every Java program that you use will make use of objects. But there are a few other kinds of programming language Things that do not have the same complex internal structure or behavior. Java, like many programming languages, has some built-in facilities for handling and manipulating simple kinds of information, like the number `42` or the character `'%'`. These Things are not objects and they cannot act on their own or

Numeric literals are `int`s...

Java Notes

Java Primitive Types

Each Java primitive type has its ow built-in name. For example, ***`int`*** is a name for a type-of-thing corresponding to an integer value. There are actually four Java names for integers, depending on how much space the computer uses to store them. An **`int`** uses 32 ***bits***, or **b**inary dig**it**s. It can represent a number between -2147483648 and 2147483647 -- from 2^{31} to 2^{31}- 1 -- which is big enough for most purposes. An integral number (i.e., a number without a decimal point) appearing literally i a Java program will be interpreted as an `int`.

If you need a larger range of numbers, you can use the Java type ***`long`***, which can hold values between - 2^{63} and 2^{63} - 1. You car just type in a value like `80951151051778`, though. Literals intended be interpreted as `long` end with the character`L` (or `l`):

be asked to provide services. They *do* have types and can be named, but their names do not behave exactly like object names. Because they are simple, these types are called ***primitives***.

Java object types may have complex internal state or the ability to perform interesting behaviors. Java primitive objects do not have any internal state, nor can they do anything by themselves. They cannot ring like an alarm clock, close like a window, or be selected like a radio button. They cannot even add themselves or display themselves on a screen. Only objects can be asked to perform actions. In chapter 5, we will learn how we can use primitive Things to accomplish useful tasks. But, unlike object Things, primitive Things cannot accomplish anything by themselves.

...or `double`s.

3.5.1 Primitive Types

A type is Java's way of indicating what kind of Thing something is and what it can do. Like objects, Java primitive Things have types. But unlike objects, you cannot create any new Java primitive types. Java has exactly eight primitive types. These types are built into the Java language, so they are always available to you. Four of these types correspond to integers. Two of the types correspond to decimal numbers. One of the types is for single characters. The eighth type is for true-or-false values, or booleans. These are all and exactly the primitive types permitted in Java. The name of each of Java's primitive types begins with a lower case letter.

For example, Java knows about numbers. If you type `6` in a(n appropriate place in a) Java program, the computer will "understand" that you are referring to an integer greater than 5 and less than 7. The expression `6`is a Java ***literal***: an expression

`80951151051778L`. There are also two smaller integer types: the 16-bit ***`short`***and the 8-bit ***`byte`***. There are no `short` or `byte` literals. Fo most purposes, the **`int`** is probably the Java integral type of choice.

WARNING: The limited range of all the integral types means that calculations using such numbers can cause errors or giv incorrect results if they would require going beyond the ranges of t types involved.. The programmer must be aware of such limits.

Real valued numbers are represented using ***floating point***notation. There are two versions of real numbers, again corresponding to the amount of space that the computer uses to store them. One is ***`float`***, short for floating point; the other is ***`double`***, for **double precision** floating point. Both are only approximations to real numbers, and double is a better approximation than float. Neither is precise enough for certain scientific calculations. A **`float`**is 32 bits. The biggest float is about 3.4×10^{38}; the smallest is about -3.4×10^{38}. The **`float`** type can represent numbers to an accuracy of about 8 significant decimal digits.

A **`double`** is 64 bits. The biggest **`double`** is about 1.8×10^{308}; the smallest is about -1.8×10^{308}. The **`double`** type can represent numbers to an accuracy of about 16 significant decimal digits.

The **`double`** type gives more precise representation of numbers (as well as a larger range), and so is more appropriate for scientific calculations. However, since erro are magnified when calculations are performed, computations with large numbers of calculations mean that unless you are careful, the imprecision inherent in these approximations will lead to large accumulated errors. [[Footnote: These issues are studied by the field of mathematics known as numerical analysis.]]

The default floating point literal is interpreted as a `double`; a literal to be treated as a `float` must end with `f` or `F`. (A `double` liter optionally ends with `d` or `D`.)

Floating point numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., `9.87E-65`or`3.e4`).

Character literals written inside single quotes: `'x'`.

The Java character type is called ***`char`***. Java characters are represented using an encoding called ***unicode***, which is an extension of the ***ascii*** encoding. Ascii encodes English alphanumeric characters as well as other characters used by American computers using 8 binary digits. Unicode is a 16-bit representation that allows encoding of most of the world's alphabe Character literals are enclosed in single quotation marks: `'x'`.

Characters that cannot easily be typed can be specified usin a ***character escape***: a backslash followed by a special character or number indicating the desired character. For example, the horizont tab character can be specified `'\t'`; newline is `'\n'`; the single quo character is `'\''`, double quote is `'\"'`, and backslash is `'\\'`. Characters can also be specified by using their unicode numeric equivalent prefixed with the `\u` escape.

Boolean literals are `true` and `false`.

The true-or-false type is called ***`boolean`***. There are exactly two `boolean` literals,`true` and `false`.

whose value is directly "understood" by the computer. In addition to integers, Java recognizes literals that approximate real numbers expressed in decimal notation as well as single textual characters.

All of the following are legitimate Things to say in Java:

- `6`
- `42`
- `3.5`
- `-3598.43101`

The names of Java primitive types are entirely lower case.

String literals written inside double quotes: `"a String!"`.

Th double-quoted-sequence-of-characters type is called ***`String`***. `String` doesn't actually belong this list because, unlike the other type listed here, `String` is not a primitive type. Note that its name begins with an upper case letter. String does have a literal representation, though. (String is the only non-primitive Java type to have a literal representation.) A `String`literal is enclosed in double quotation marks: `"What a String!"` It may contain any character permitted in a character literal, including the character escapes described above. The `String` `"Hello, world!\n"` ends with a newline.

The names of Java primitive types are reserved words in Java. This means that they have special meanings and cannot be use to name other Things in Java. (See the sidebar on Java Names.)

Details of Java numeric literals -- and of all of the other literals discussed here -- are covered in the sidebar on Java Primitive Types. As we will see in chapter 5, you can perform all of the usual arithmetic operations with Java's numbers. [[Footnote: Be warned, though, that non-integral values -- like 1/3 and 1.234567890123456789 -- are, in general, represented only approximately.]]

Java can also manipulate letters and other ***characters***. When you type them into Java, you have to surround each character with a pair of single quotation marks: `'a'`, `'x'`, or `'%'`, for example. Note that this enables Java to tell the difference between `6` (the integer between 5 and 7) and `'6'`(the character 6, which on my keyboard is a lower case '^'). The first is something that you can add or subtract. The second is not.

One character by itself is not often very useful, so Java can also manipulate sequences of characters called ***strings***. Strings are used, for example, to communicate with the user. Error message, user input (i.e., what you type to a running Java program), titles and captions are all examples of Java strings. To describe a specific string in Java -- for example, the message that your computer prints to the screen when you boot it up -- you can write it out surrounded by double quotes: `"Hi, how are you?"` or `"#^$%&&*%^$"`or even `"2 + 2"`. Your computer doesn't *understand* the string, it just remembers it. (For example, the computer doesn't know of any particular relationship between the last example and the number `4` -- or the string `"4"`.)

It turns out that it's also useful for many programs to be able to manipulate conditions, too, so Java has one last kind of primitive value. For example, if we are making sandwiches, it might be important to represent whether we've run out of bread. We can talk about what to do when the bread basket is empty:

if the bread basket is empty, buy some more bread....

Conditions like this -- bread-basket emptiness -- are either true or false. We call this kind of Thing a ***boolean*** value. Booleans are almost always used in conditional -- or test -- statements to determine flow of control, i.e., what should this piece of the program do next? Java recognizes ***`true`***and ***`false`*** as boolean literals:

if you type one of them in an appropriate place in your program, Java will treat it as the corresponding truth value.

As we have just seen, primitive-type Things can be referenced by literals. In addition, Strings can be referenced by literals, by using the double-quotation mark syntax: "What, me worry?" So all three of the following are literals -- "5" '5' and 5 -- but the first is an object-type Thing (a String), while the latter two are primitive-type Things (a char and an int, respectively).

There are a lot of rules about how these different Things work and how they are used. For details on Java's primitive types -- including their names and their properties, see the sidebar on Java Primitive Types.

3.5.2 Primitive Names are Different

If you want to refer to a Java object, you can do so using an appropriately typed name. You can also refer to a Java primitive Thing using a name. Declaration, assignment, and definition are used with primitive names in much the same way that they work with object names. For example:

```
int myLuckyNumber = 6;

int yourGuess;

yourGuess = myLuckyNumber;
```

The first line creates a new name, `myLuckyNumber`, and binds it to 6. The second line creates a new name, `yourGuess`, but does not give it any initial value. [[Footnote: Whether yourGuess is given an initial value by this line depends at least on where the line appears within code and in some cases on the particular implementation of Java in which you are executing. It is *always* inadvisable to assume that yourGuess has been given a value, though -- being a dial -- it must, of course, be set to some value.]] The final line assigns the value of `myLuckyNumber`-- 6 -- to the name `yourGuess`.

So far, names with primitive type look a lot like object names. But it turns out that there are some important, if subtle, differences. Java names with primitive types aren't exactly labels, as object names are. You see, there may be an unpredictably large number of Buttons or KlingonStarships, and so a label is the best way to keep track of any particular KlingonStarship that comes along. But Java primitives are different.

For example, there are only two boolean values possible in Java: true and false. This means that we can use a very different mechanism for a name that has type boolean. Java does just that. A name that can have only one or the other of two values can be represented using a switch: It can be on, or it can be off. This is just what Java does: **Boolean names are just switches.** Let's say we have a boolean name, `isSunny`. This name is just a switch. It is always set in one of the two positions: on/true, or off/false. So if the switch corresponding to `isSunny` is on, we'll know that `isSunny`is true. If the isSunny switch is off, we'll know that isSunny is false. We can tell the value corresponding to the boolean name just by inspecting the switch.

But if we use a switch to indicate true-or-false, how do we do assignment? When we use label names -- for objects -- we just stick a second label on the same object. When we use switches as names, creating a new name means creating a new switch. That is, **dial name assignment copies one dial's setting to another**. So what do we do when, for example, we have

```
boolean amHappy = isSunny
```

(which means that I am happy if it's sunny, and not happy if it's not)? Simple enough: we set the new switch -- amHappy -- to the same position that isSunny is in. (Note: We do this once, at the time of the assignment. After that, the two switches are completely separate. More on this later....)

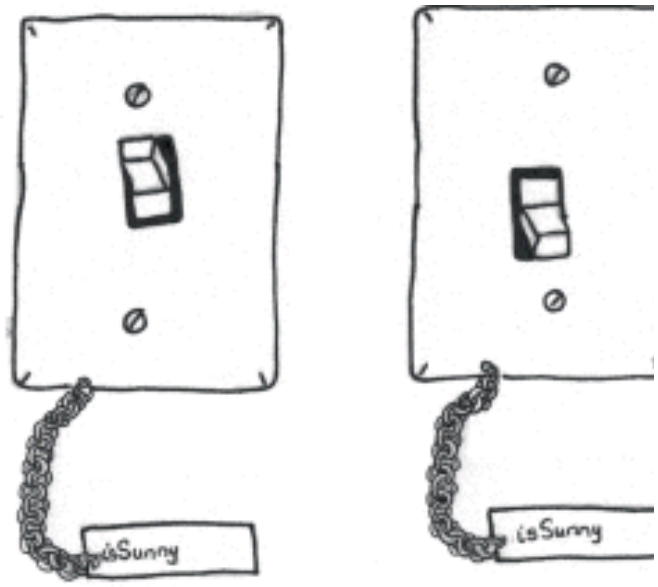


Figure 3.8. A boolean name is just a switch.

It turns out that this analogy works for all of the Java primitive types. For example, there are only a fixed, finite number of ints possible in Java. (See sidebar.) Although it might be confusing to imagine a switch with 4,294,967,296 positions, you can imagine that an int name is just a very large dial with those same 4 billion settings. By reading the setting of the dial, you can tell what value an int name corresponds to.

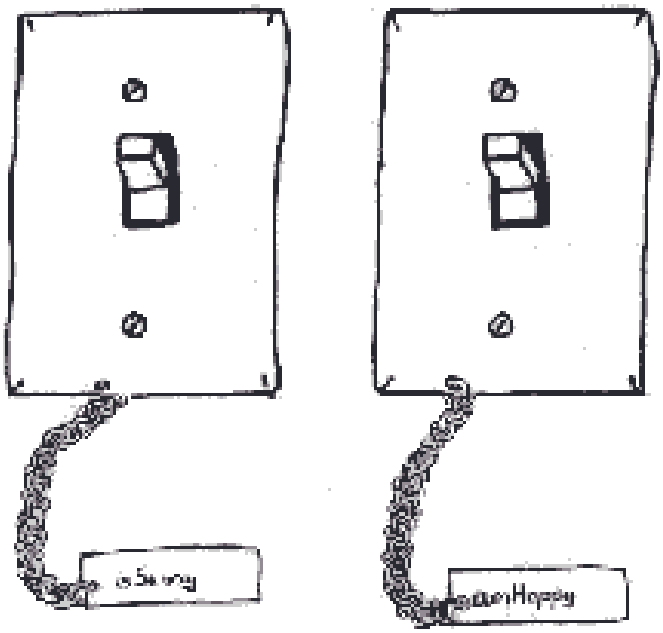


Figure 3.9. Copying values from one switch to another.

As remarkable as it may seem, each of the Java primitive types is represented in this way. The dial here is metaphorical, but the actual representation is very much as described. A Java primitive is stored in such a way that every name with a primitive type indicates its value just as the metaphoric dial does. A name corresponding to a `byte` is simply a dial with 256 positions. A `long` name has 2^{64} positions. And even `float` and `double` names have only finite numbers of positions; this is why floating point numbers don't *really* represent real numbers and in fact aren't all that good for extreme precision calculations. [[Footnote: Even a double precision floating point number can only represent 2^{64} values between -1.8e ^{308} and 1.8e ^{308}, so many values just can't be captured accurately. Note also that the actual precision of a double varies over this range.]] `Char` ranges

over many more values than just a-Z, but there are still only 2^{16}possible characters in Java, so a char name is like a dial with that many positions.

For example,

```
int i;
```

associates `i` with a dial that's just the right size for a 32-bit integer.

3.5.3 Dials and Labels are Different

How is this different from a label? There are at least three big differences.

1. **Labels can be `null`; dials cannot.** When a label is created, it may not be attached to anything. A dial can't be created without having a value; by its very nature, *the dial's hand is on **some** value.* [[Footnote: In fact, the declaration of a dial-type name not only sets up an appropriately sized dial, it also sets that dial. This means that you must give a name a value before you can use it. Some special kinds of names get values by default. We will mention these values as the names are introduced. In fact, this isn't just true when it's created. A label can be unstuck -- null -- but *a dial always has **exactly one** value.*]]
2. **When a type X label is declared, no object of type X is created; when a type Y dial is declared, the dial is created and has a value.** When a label of a certain object-type is declared, a label name appropriate for that object-type is created (and storage allocated for it), but no object of that object-type is created. When a dial of a certain primitive-type is declared, a dial name appropriate for that primitive-type is created and also, by the very nature of a dial, its hand is set to *some* value of that primitive-type. [[Footnote: Note, however, that it may be illegal to access this "default" value; you should always be sure that a name has been given a value before you access it. You will learn more about the values initially given names as you learn about different kinds of names in Java.]]
3. **Labels can share a value; dials cannot.** Two labels can be stuck on to the same object; then, changing the object by using the first label to reference it changes the (same) object referenced by the second label. While two dials can have the same value (both are set to the same position on the dial), the dials are independent -- changing the setting on one dial never changes the setting on another dial.

   In particular, assigning one label to another means that both are stuck on the same object (or both are null), so that they "share" a value. When one dial is assigned to another, the setting on the latter dial is copied onto the former. That's it; after that, the assignment is complete and the two dials go their separate ways. There is no further relationship between the values on the dials.

The last point is worth a closer look. Consider the following two very similar-looking sets of statements (one on the left, the other on the right):

```
int i;            Cat marigold;
```

```
i = 3;              marigold = new Cat();

int j = i;          Cat phoebe = marigold;

i = 4               marigold.haveKittens();
```

In both cases, the first statement declares a name and the second statement assigns the name a value. The name `i` on the left is a dial-name and is set to 3; the name `marigold` on the right is a label-name and is set to a new `Cat`.

In both cases, the third statement declares another name and assigns the first name to the second. However, assignment has a different meaning in these two cases. On the left, the dial-name `j`is set to the same setting (3) as the dial-name `i`. On the right, the label-name `phoebe` is stuck onto the same object that the label-name `marigold` is stuck on.

In both cases, the fourth statement changes something by using the first name (`i` on the left and `marigold`; on the right). However, the effects are quite different. On the left, the value of `i` changes to 4, of course, but the value of `j` remains 3. If we asked whether`j` is 4, the answer would be "No." On the right, the object referenced by `marigold` has kittens, so that (same) object referenced by`phoebe` has kittens. If we asked whether `phoebe` has had kittens, the answer would be "Yes!"

The left side of the figure shows the *dial* names `i` and `j`. The right side of the figure shows the *label* names `Marigold` and `Phoebe`. The middle row shows the different effects of assignment in the two cases.

To summarize:

- When you are using names that are declared to be of object-types, think of those names as labels that can be stuck on objects of the declared type.
- When you are using names that are declared to be of primitive-types, think of those names as dials that always are set to some physical point.
- This implies three key distinctions:
  - *Labels can be `null`; dials cannot.*
  - *Declaring a label does NOT create an object of the declared type.*
  - *Labels can "share" a value; dials cannot.*

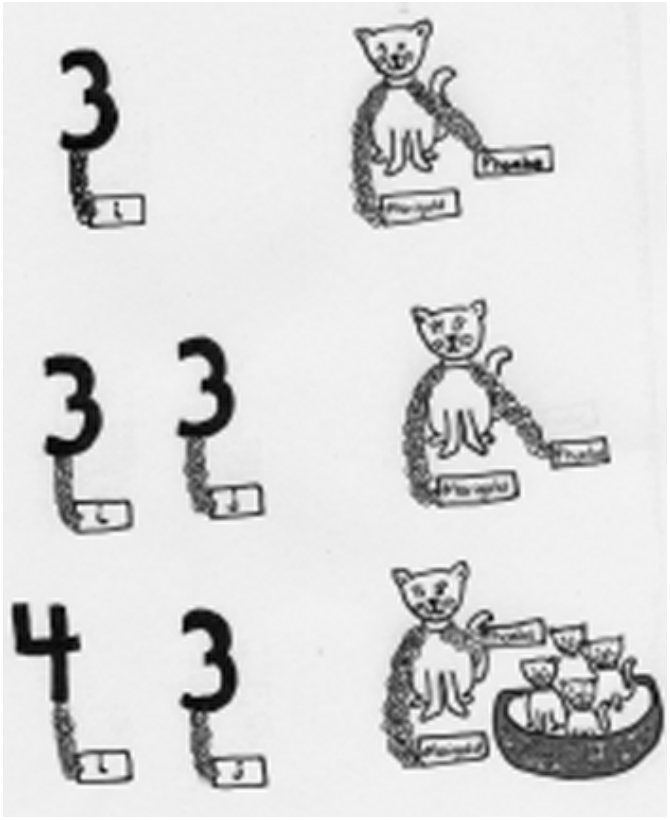


Figure 3.10. Dial names versus label names: the effect of assignment.

- It is easy to recognize primitive-types: they are all lower-case letters, while (by convention) we always use an upper-case letter as the first letter of any object-type.

  A more formal term for dial types is ***value type***.

Chapter Summary

- In Java, most Things are objects. A object's type tells you what that object can do.
- Generally, it is an instance of a type, rather than the type itself, that provides behavior. You can make a request of an object using the . () syntax:

  ```
  object.service()
  ```

  asks *object* to perform *service*.
- `String` is the type for arbitrary text. `String` is the only Java object type with literals.
- `Console` is a cs101 library object that allows you to communicate with a user using Strings.
- A Thing can be accessed using a name or an indirect reference. If you do not hold onto a Thing -- using a name -- it may be dropped on the floor and you will only be able to access it through an indirect reference, if at all.
- Names can be used as placeholders for values. Every name is born (declared) with a particular type, and can only label Things having that type. A name can be associated with a value through assignment. Definition combines declaration with assignment.
- Object names are labels.
  - Declaring an object name does not create an object of the declared type. An object name can be null (not stuck on anything).
  - An object name can be null (not stuck on anything).
  - Assigning an object name simply sticks that label on the object. Object name assignments do not create new copies of objects.
  - Two object names can label (be stuck on) the same object, sharing it as a value. If you change the object, the change will be visible through either name as long as they share that value.
- Java has eight primitive types:
  - `char` is the type for single keystrokes (letters, numbers, etc.)
  - `int` is the standard type for integers. Other integer types include `byte`, `short`, and `long`.

- `double` is the standard type for floating point numbers, which are approximations to real numbers. `float` is another floating-point type.
- `boolean` is a type with only two values, `true` and `false`.
- The limited range of all the numeric types means that calculations using such numbers can errors or incorrect results when they are used outside of their ranges. In addition, the limited precision of the floating-point types means that repeated calculations involved them can lead to large accumulated errors. The programmer must be aware of such differences between computer arithmetic and real arithmetic.
- The Java keyword for each of the eight of primitive types begins with a lower-case letter.

All other Java types are object types. By convention, we begin each object type with an upper-case letter.

- Primitive types have dial names. A dial name always has an associated value. Two dials cannot share a single value; each has its own copy.
- Literals are Things you can type directly to Java. Only the primitive types and the single object type `String` have literals.

Exercises

1. Assume that the following declarations apply:

```
int i; char c; boolean b;
```

For each item below, give the type of the item.

a. `42`

b. `-7.343`

c. `i`

d. `'c'`

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

f. `b`

g. `false`

h. `"false"`

i. `c`

j. `'b'`

k. `"b"`

2. For each of the following definitions, fill in a type that would make the assignment legal.

```
__________ a = 3;
__________ b = true;
__________ c = 3.5;
__________ d = "true";
__________ e = "6";
__________ f = null;
__________ g = 0;
__________ h = '3';
__________ i = '\n';
__________ j = "\n";
```

[[Footnote: There are several answers to some of these, but in each case only one "most obvious" type. It is this "most obvious" type that we are after.]]

3. This problem checks your understanding of *assignment*.

   a. Assume that the following statements are executed, in order.

```
int a = 5;
int b = 7;
int c = 3;
int d = 0;
a = b;
c = d;
```

```
a = d;
```

What is the value of a? of b? of c? of d?

b. Assume that the following statements are executed, in order.

```
int a = 5;

int b = 7;

int c = 3;

int d = 0;

a = b;

b = c;
```

What is the value of a? Of b? Of c?

c. Assume that the following statements are executed, in order.

```
char a = 'a';

char b = 'b';

char c = 'c';

char d = 'd';

a = b;

c = a;

a = d;
```

What is the value of a? Of b? Of c? Of d?

d. Assume that `myObject` is a name bound to an object (i.e.,`myObject`is not `null`). After the following statements are executed in order,

```
Object a = myObject;

Object b = null;

Object c = a;

a = b;
```

Is the value of a *null* or non-*null*? What about b? What about c? What about myObject?

e. Assume again that `myObject` is a name bound to an object (i.e.,`myObject` is not `null`). After the following statements are executed in order,

```
Object d = myObject;
d = null;
```

is the value of d *null* or non-*null*? What about myObject?

f. Assume one more time that `myObject` is a name bound to an object (i.e., `myObject` is not `null`). After the following statements are executed in order,

```
Object e = myObject;
myObject = null;
```

Now is the value of of e *null* or non-*null*? What about myObject?

4. Which of the following could legitimately be used as a name in Java? (Note that none of them would be wise choices for names, except possibly in a Star Wars game, as none of them is likely to convey meaningful information to readers of a program.)

```
3PO
R2D2
c3po
luke
jabba_the_hut
PrincessLeia
Han Solo
obi-wan
foo
int
Double
character
string
goto
elseif
fi
```

5. Assume that the following declarations have been made:

```
int i = 3;

int j;

char c = '?';

char d = '\n';

boolean b;

String s = "A literal";

String s2;

Object o;
```

Complete the following table:

Name	dial or label?	Value (or null?)
i		
j		
c		
d		
b		
s		
s2		
o		

6. In the section on names as labels, we put a label on the fortune teller's fortune:

```
myPersonalFortune = fortuneTeller.getFortune()
```

Assume that this fortune is "You will live a long and happy life." Now assume that the fortune teller is asked for another fortune:

```
fortuneTeller.getFortune()
```

Note that this fortune -- assume that it is "Things change." -- is not assigned explicitly to any label. At this point, we execute the fortune-printing statements from the same section:

```
Console.print( "And your fortune is...." );

Console.println( myPersonalFortune );
```

What is printed?

7. Assume that there is an already -defined object type called*Date* and that *today* is an already -defined Date name with a value representing today's date. Suppose that you wanted to declare a new name, *yesterday*, and give it the value currently referred to by *today*. This would be useful, for example, if it were nearly midnight and we might soon want to update the value referred to by *today*.

   Explain why the following attempt will *not* successfully solve this problem.

   ```
   Date yesterday;

   yesterday = today;
   ```

   [[Footnote: At this point of this book, you should understand why the above will *not* work, but we have not yet discussed what *would* work. We'll see the basic ideas for a successful solution in Chapter 7,*Building New Things: Classes and Objects*. Also relevant is the discussion about @@ *clone* and Cloneable in Chapter 10, *Inheritance*.]]

8. Continuing the previous problem, now assume that *today* is an already -defined `int` (not Date) name with a value representing today's date, where 1 represents January 1, and 32 represents February 1, and 33 represents February 2, and so on, with 365 representing December 31. (Assume that this is not a leap year.) Again suppose that you wanted to declare a new name,*yesterday*, and give it the value currently referred to by *today*. Now, the problem is solvable with the tools from this chapter.

   Give the solution (that is, declare *yesterday* appropriately and give it the value referred to by *today*). Then explain how this problem is different from the previous problem.

9. (From the footnote): In the tale of two hats, what would have happened if the King had executed his hat-checking lines in the opposite order?

   ```
   kingGeorgeHat = null;

   hatCheck.checkHat( kingGeorgeHat );
   ```

10. Recall again the tale of two hats. Later, after the events of that story were long past, someone suggested that the problem here was that the hat check hadn't issued claim checks, which might have changed the circumstances. These claim checks would have been `ints`, not `Hats`.

    "If the names had been dials rather than labels, would Mrs. Roosevelt's assistance still have been needed?"

    As it happens, there *were* claim checks involved, but Charlie Chaplin and King George had clumsily dropped them, and they were unable to determine which claim check was whose. This is why they'd each used the `hatCheck.returnHat()` service, rather than a service that required a claim check as input.

Suppose that the situation were as follows:

After the mixup, Mr. Chaplin found himself holding claim check 2:

```
int charlieChaplinCheck = 2;
```

while King George was in possession of claim check 1:

```
int kingGeorgeCheck = 1;
```

However, the crown can only be retrieved with claim check 2, the bowler with check 1. So the two gentlemen are now faced with a swap of integers rather than hats. As the story originally unfolded, they gave up and later were forced to proceed with the hat swap. Imagine, instead, that Mrs. Roosevelt had walked up at this moment -- when the claim checks needed rearranging -- and write code to resolve the situation WITHOUT using the literals 1 and 2 further. (You may, of course, use `charlieChaplinCheck` and `kingGeorgeCheck` in your code.)

Questions or comments:
<webmaster@cs101.org>