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. Primitive things can't do anything by themselves, but in later chapters you'll learn how to do things with them. 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.

The remainder of this chapter introduces two important concepts for understanding and manipulating things in Java: typing and naming.

Types are ways of looking at things. 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. 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.

Names are ways of referring to things that already exist. 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, the type describes how you can interact with the thing that the name names. There are actually two different kinds of names in Java: primitive (shoebox) 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 this Chapter**

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

What kinds of things can your programs involve? Almost anything, as it turns out. But we'll start with some very simple things.

3.1.1 Primitive Things and Literals

Java, like many programming languages, has some built-in facilities for handling and manipulating simple kinds of information. 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. 6 is a Java **literal**: an expression whose value is directly "understood" literally by the computer. In addition to integers, Java recognizes literals that approximate real numbers expressed in decimal notation as well as single textual characters.

This means that all of the following are legitimate things to say in Java:

- 6
- 42
- 3.5
- -3598.43101

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 the next chapter, you can perform all of the usual arithmetic operations with Java's numbers.

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 '%'. 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 'a'). 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

---

1 Be warned, though, that non-integer values -- real numbers -- are represented only approximately.
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.

There are lots of rules about how these different things work and how they are used. For many of the detailed rules about the primitive things that we have just covered, see the sidebar on Java Primitive Types.

### 3.1.2 Objects

The things described above are very specific kinds of things, and they have very limited functionality. In the next chapter, we will see what we can do to manipulate these primitive kinds of things. Most of what goes on in Java, though, concerns another kind of thing. This kind of thing can include anything you might want to represent in a computer program. Some examples of these other things 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. These things -- everything else your program can talk about -- are called objects. In Java, objects include everything that is not one of the aforementioned primitive types.²

² In fact, in Java, strings are objects and not primitives.
There are many different kinds of objects, from buttons and windows to dictionaries and factories. Each kind of object has a type associated with it. Objects can be asked to do things, and 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. 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.

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.

As you can see, objects can be much more interesting than the kinds of things represented by Java primitive types. However, objects are somewhat more complex than Java primitives. In particular, there are no object literals: you can't type an arbitrary object directly into your program the way that you can type 3 or 'x' or "Hello!" or false.

Almost everything that you do in Java uses objects, and you will hear much more about them throughout this book. This chapter concentrates on how you identify the Things in a program and how names can be used to refer to them. In the next chapter, we will see in more detail how to use these Things to produce other Things. Chapter 5 concentrates on combining these pieces into a full-blown recipe, a single list of instructions that can be followed to accomplish a particular job. The three chapters following (6-8) look at objects in more detail, describing how to create and use the objects that are manipulated by these instructions, and how these instructions themselves can be combined to form objects and entities that interact in a community.

3 Except String literals.
3.2 Naming 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 Java offers for keeping 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 or putting the thing in a particular shoebox. (We'll see later that there are actually two different kinds of name/thing relationships, one more like labels and the other more like shoeboxes.) We sometimes say that the name is bound to that value.
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 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   default   if   private   throw
- boolean    do        implements protected throws
- break      double    import    public  transient
- byte       else       instanceof return  try
- case       extends   int       short   void
- catch       final      interface static  volatile
- char       finally    long      super   while
- class       float      native    switch
- const       for        new       synchronized
- continue    goto       package   this

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. -- do 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.

To actually assign a value to a name -- to create a binding between that name and that value -- Java uses the syntax

\[ name = value \]

For example,
myFavoriteNumber = 4

This associates the value 4 with the name myFavoriteNumber. 4 may be associated with more than one name, but 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 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".\(^4\))

Once a particular name refers to a particular thing -- say greeting has the value "Hi, how are you?" -- then we can use the name wherever we would use its value, with the same effect. The name becomes a stand-in for the thing it refers to. In the next chapter, we will see that a name is a simple kind of expression. But before we can assign a value to a name, we need to know whether the name is allowed to label values of that type.

### 3.3 Types

Up to now, we've been pretty casual about our things. Java, however, is a strongly typed language, meaning that it is not at all casual about what kind of thing something is. Each Java thing comes into the world with a type, i.e., an indication of what kind of thing it is. Java names, too, are created with types, and a Java name can only be used to label objects of the appropriate type. Before we can use a name -- as myFavoriteNumber, above -- we have to declare it to be of a particular type. Declaring a name means stating that that particular name is to be used for labeling values (things, objects) of some particular type.

#### 3.3.1 Declarations and the type-of-thing name-of-thing rule

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

```java
int myFavoriteNumber;
char firstLetterOfMyName;
```

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, myFavoriteNumber is being declared to have type int. This is a Java name for an integer. Finally, each declaration ends with a semi-colon (;). So

---

\(^4\) Barring weird interpersonal pileups, of course.
the first declaration here creates a name, `myFavoriteNumber`, suitable for naming integers (or, in Java, `ints`). The second line creates the name `firstLetterOfMyName`, suitable for naming single characters (i.e., things of Java type `char`).

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. (For example, `myFavoriteNumber` may initially be 4, but later change to be 13.) The association between a name and a type persists for the lifetime of the name, however. (`myFavoriteNumber` can only name an `int`, not a `String` or a `boolean`.)

### 3.3.2 Definition = Declaration + Assignment

Declaring a name begins its useful lifetime. 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 at the time that it is declared. This combination of a declaration and an assignment is called a **definition**. (Declarations tell you what type is associated with a name. Assignments tell you what value that name is bound to. In fact, assignments set the values of names. Definitions combine the "what kind of thing it can name" and "what value it has" statement types.) For example:

```java
    boolean isHappy = true;
    double degreesCelsius = 0.0;
    Thread spirit = new Thread(this);
    Cat myPet = marigold;
```

The first and second of these make use of `boolean` and `double` constants, respectively, to assign values to the names `isHappy` and `degreesCelsius`. The `Thread` definition actually creates a new `Thread` using a constructor with one argument; much more on this later. 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, `myPet` is bound to the actual `Cat`, not to the name. If `marigold` later refers to some other `Cat` -- say both `Cats` undergo name changes -- `myPet` will still refer to the `Cat` originally known as `marigold`. 
3.3.3 Primitive Types

A type tells Java something about how it should represent and manipulate the information internally. All of the Java types discussed above except Cat have built-in type names. These type names are a part of the Java language. For example, characters -- such as 'x', '3', ';', or '#' -- have type char. The second line of the first example above shows the Java type for a character -- char -- and declares that firstLetterOfMyName is a name that can be used to refer to a character. Each Java type also has associated representational properties. All of the Java primitive type names, along with their properties, are described in the sidebar on Java Primitive Types.
Java Primitive Types

Each Java primitive type has its own 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 binary digits. It can represent a number between \(-2^{31}\) and \(2^{31} - 1\) -- which is big enough for most purposes. An integral number (i.e., a number without a decimal point) appearing literally in 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 can't just type in a value like 80951151051778, though. Literals intended to be interpreted as long end with the character L (or l): 80951151051778L. There are also two smaller integer types: the 16-bit short and the 8-bit byte. There are no short or byte literals. For most purposes, the int is probably the Java integral type of choice.

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, from positive or negative \(3.4e^{38}\) to +/-1.4e\(-45\); a double is 64 bits, from \(1.8e^{308}\) to \(4.9e^{324}\). The double type gives more precise representations of numbers (as well as a larger range), and so is more appropriate for scientific calculations. However, since errors 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.\(^5\)

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 literal optionally ends with d or D.)

You can express both integral and real number literals with or without a

\(^5\) These issues are studied by the field of mathematics known as numerical analysis.
Real and rational numbers can be written using decimal notation, as in the text, or in scientific notation (e.g., 9.87E-65 or 3.e4).

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 alphabets. Character literals are enclosed in single quotation marks: `'x'`.

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

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

The names of Java primitive types are entirely lower case.

The double-quoted-sequence-of-characters type is called `String`. `String` doesn't actually belong in 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 used to name other things in Java. (See the sidebar on Java Names.)

### 3.3.4 Object Types

Java also comes with certain predefined object types, such as `String` and `Button`. If you are using the cs101 course libraries, you'll also have access to object types
such as AnimateObject and DefaultFrame. And, 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 or of your own definition -- are all kinds of objects. Note that, by convention, the name of each object type -- each class -- starts with a capital letter. The names of the primitive types start with lower case letters, as do (most) names and methods.

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). Each of these object types may describe many different individual objects -- the three KlingonStarship\$ 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 individual objects are sometimes called instances of their types. 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'll explore this idea in greater detail in chapter 7.)

Each individual object comes ready-made with certain properties and behavior. 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 do things (including telling us about themselves) using specific services that these objects provide. Often, these services are accessed 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". These requests are the most basic form of interaction among the entities in our community.

One particularly useful object is Console. Console is an object that can print a String to the Java console, a standard place where someone running a Java program can look for information. Console can also readln a String that the user types to the Java console.
Console

Console is a special cs101 object that knows how to communicate with the user in some very basic ways. If your program says

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

then the String "Hello there!" will appear on the Java console. The command

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

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

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

would produce the output

A is for apple.

while

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

would produce

A

is for apple.

You can of course combine `prints` and `printlns` arbitrarily. Printing a String containing a newline character escape (`\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 has one other important method, `Console.readLine()`, 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.

3.4 Types of Names

In Java, every name has a type. This type is associated with the name when the
name is declared. *The type associated with a particular name never changes.* It turns out that there are two rather different kinds of names in Java. In this section, we will look at each in turn and see what it means for a name to be declared to be of a particular type.

### 3.4.1 Shoebox Names

Names, in Java, come in two flavors. The first kind of name is rather like a shoebox. Declaring the name to be of that type creates a space in the computer just the right shape and size to hold the appropriate thing.

For example,

```java
int i;
```

associates `i` with storage appropriate for a 32-bit integer. In fact, the declaration of a shoebox-type name not only sets up an appropriately sized shoebox, it also fills that shoebox with an appropriate value. If -- as in this declaration of `i` -- no value is specified, the shoebox contains the default value for the type -- in this case, 0. *There is no such thing as an empty shoebox.* You must give a name a value before you can use it.

Assigning a value to such a name replaces the value stored in the shoebox with a new copy of the appropriate value. *There is no sharing between shoeboxes.* Instead, there are multiple copies of, say, the int 3. *Every shoebox always contains exactly one thing.* When a new value is assigned to a shoebox, any value previously stored in that shoebox is discarded. So, for example,

```java
i = 3;
```

---

6 Some special kinds of names get values by default. We will mention these values as the names are introduced.
makes the i-shoebox hold 3; the 0 initially stored in the i-shoebox is discarded.

The declaration-plus-assignment definition

```c
int j = i;
```

creates another int-sized shoebox, j. In this case -- because this is a definition, not simply a declaration -- j starts out containing a copy of the value that happens to be in i when the definition is executed. That is, this definition makes a copy of the value currently in i -- 3 -- and creates a new shoebox, called j, to hold it. Once this is done, there is no special relationship between i and j.

In particular, if we now change the value of i:

```c
i = 4;
```

-- which sets the value of i to 4 -- j is unchanged; it still holds 3.

At any point in time, each shoebox contains exactly one thing. A shoebox cannot be empty. A shoebox also cannot contain more than one thing. If you put a new thing into a shoebox, the thing that was previously there is no longer there.

Strictly speaking, the kinds of things that go into shoeboxes are things that are the same exactly when they look the same. For example, two "different" copies of the (int-sized) number 3 are, for all intents and purposes, the same.7

This might make more sense if contrasted with two things that "look" the same, but aren't. Consider, for example, identical twins. Although they may look exactly

---

7 Note, however, that this does not extend to 3 and 3.0 and 3.0f, each of which is a different thing. This is because each of these has a different type.
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. 3, on the other hand, 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.

Notice, though, that although 3 is 3 is 3 (i.e., there aren't "different" 3s the way that there are different twins), there may be different shoeboxes that CONTAIN 3. If myBox and yourBox are both int-sized shoeboxes, each containing 3, changing the number in myBox doesn't automatically change the number in yourBox. So after

```javascript
int myBox = 3;
int yourBox = 3;
myBox = 5;
```

yourBox will still contain 3. Each shoebox is separate and (unless we find some way to actively connect it to another) independent.

The only thing that remains to say about shoebox-type names is how to recognize one. The rule is quite simple: All names with primitive type are shoebox-type names.

A more formal term for shoebox types is value type.

![Figure 4. Label names.](image)

3.4.2 Label Names

We have seen that names associated with primitive types are shoebox-type names. Names associated with all other types -- including all Object types -- are label or reference names. (This includes String names.)

Label-type names are names that can be stuck onto (appropriately typed) objects. 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

```java
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

```java
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`.)

After we give `myButton` a value -- stick it onto a particular `RadioButton` -- we can check to see whether it's pressed:
myButton.isSelected()

(This is an expression that returns a boolean value; see the discussion of expressions in the next chapter.)

If we now declare

RadioButton yourButton = myButton;

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.

![Diagram showing two labels sharing a reference](image)

Figure 6. Multiple labels can refer to the same object.

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()

would 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.

All non-primitive types work like labels.

Chapter Summary

• Literals are Things you can type directly to Java.

• Java has several 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 real numbers. float is another real type.
  • boolean is a type with only two values, true and false.

• All other Java types are object types.

• String is the type for arbitrary text. String is not a primitive type, but Java does have String literals.
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.

Primitive types have shoebox names. A shoebox name always has an associated value. Two shoeboxes cannot share a single value; each has its own copy.

Object types have label names. Two label names can label the same object. A label that is not currently stuck on anything is associated with the non-value null.

Exercises

1. Assume that the following declarations apply:

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

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

1. 42
2. -7.343
3. i
4. 'c'
5. "An expression in double-quotes"
6. b
7. false
8. "false"
9. c
10. 'b'
11. "b"
2. For each of the following definitions, fill in a type that would make the assignment legal.\(^8\)

\[
\begin{align*}
\text{__________} & \text{ a = 3; } \\
\text{__________} & \text{ b = true; } \\
\text{__________} & \text{ c = 3.5; } \\
\text{__________} & \text{ d = "true";} \\
\text{__________} & \text{ e = "6";} \\
\text{__________} & \text{ f = null;} \\
\text{__________} & \text{ g = 0;} \\
\text{__________} & \text{ h = '3'; } \\
\text{__________} & \text{ i = '\n'; } \\
\text{__________} & \text{ j = "\n";} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{int a = 5, b = 7, c = 3, d=0;} \\
a = b; \\
c = d; \\
a = d;
\end{align*}
\]

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

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

\[
\begin{align*}
\text{int a = 5, b = 7, c = 3, d=0;} \\
a = b; \\
b = c;
\end{align*}
\]

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

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

\[
\begin{align*}
\text{char a = 'a', b = 'b', c = 'c', d='d';} \\
a = b; \\
c = a; \\
a = d;
\end{align*}
\]

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

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

\[
\begin{align*}
\text{Object a = myObject;} \\
\text{Object b = null;} \\
\text{Object c = a;}
\end{align*}
\]

\(^8\) 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.
a = b;

which of a, b, c, or myObject is null? (The answer may be none, one, or more than one.)

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 either one or both of d, myObject null?

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 which of e, myObject (or neither, or both) is null?

4. Which of the following could legitimately be used as a name in Java?

- 3PO
- R2D2
- c3po
- luke
- jabba_the_hut
- PrincessLeia
- Han Solo
- obi-wan
- foo
- int
- Double
- character
- string
- goto
- elsif
- 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>shoebox or label?</th>
<th>Value (or null?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. 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. Declare a new name, `yesterday`, and give it the value currently referred to by `today`. (This would be useful, e.g., if it were midnight and we might soon want to update the value referred to by `today`.)