References

Primitive vs. Reference Types

- Java types are divided into two different categories:
  - The built-in types are called ***primitive types***
    - Includes **`boolean`**, **`char`**, **`int`**, **`double`**
  - All other types are called ***reference types*** (or ***class types***)
    - Includes `String`, `XMLTree`, `SimpleReader`, `SimpleWriter`, `NaturalNumber`, ...

Primitive vs. Reference Types

There is no limit on the number of other ***user-defined*** types that can be developed.

- ...divided into two different
  - ...are called ***primitive types***
    - ...`ean`, ...`ar`, **`int`**, **`double`**
  - All other types are calle... ***reference types*** (or ***class types***)
    - Includes `String`, `XMLTree`, `SimpleReader`, `SimpleWriter`, `NaturalNumber`, ...

Categories of Types, v. 1

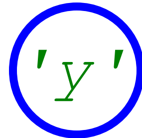
Primitive Types	Reference Types	
`boolean`	`String`	`SimpleReader`
`char`	`XMLTree`	`SimpleWriter`
`int`	`...`	`NaturalNumber`
`double`		`...`
`(plus 4 others)`		

Primitive vs. Reference Variables

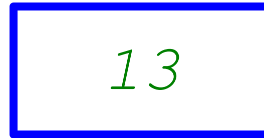
- A ***primitive variable*** is a variable of a primitive type
  - This term is used sparingly in practice, and is introduced here for parsimony to distinguish a variable of a primitive type from…
- A ***reference variable*** is a variable of a reference type
  - A reference variable is fundamentally different from a primitive variable in ways that can dramatically impact how you reason about program behavior; beware!

Examples

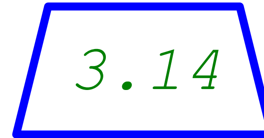
b



c



i

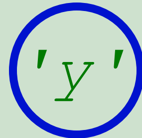


d

s

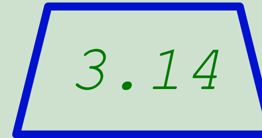
Examples

b



c

i



d

s

Recall We Said Earlier...

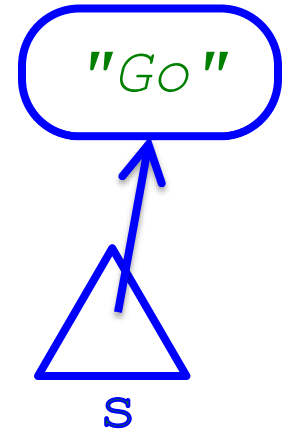
"Go"

s

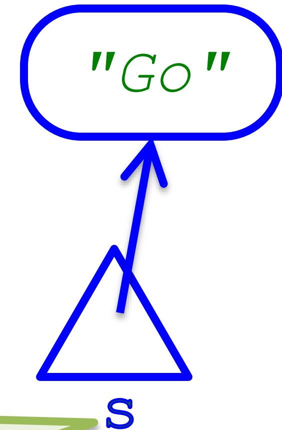
This is a `String` variable `s` whose value is "Go", i.e.,

s = "Go"

… But Here's the "Real Picture"!



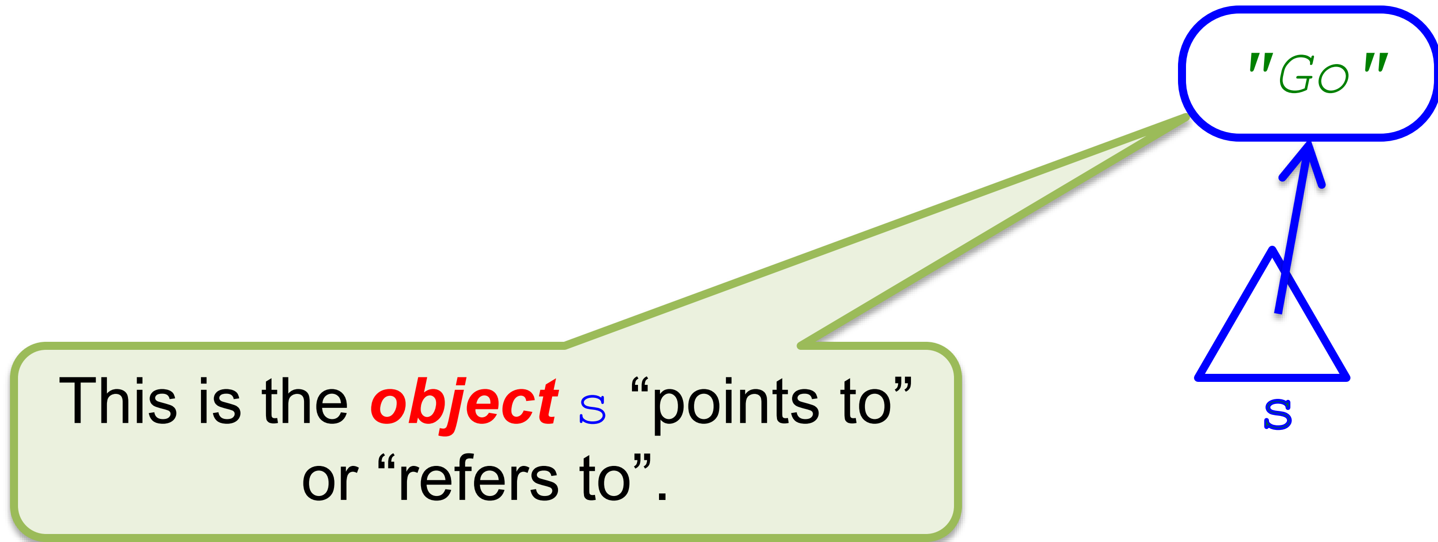
… But Here's the "Real Picture"!



There is a `String` variable `s`, whose value is a ***reference*** to an ***object*** whose value is *"Go"*.

References and Objects

References and Objects



Reference and Object Values

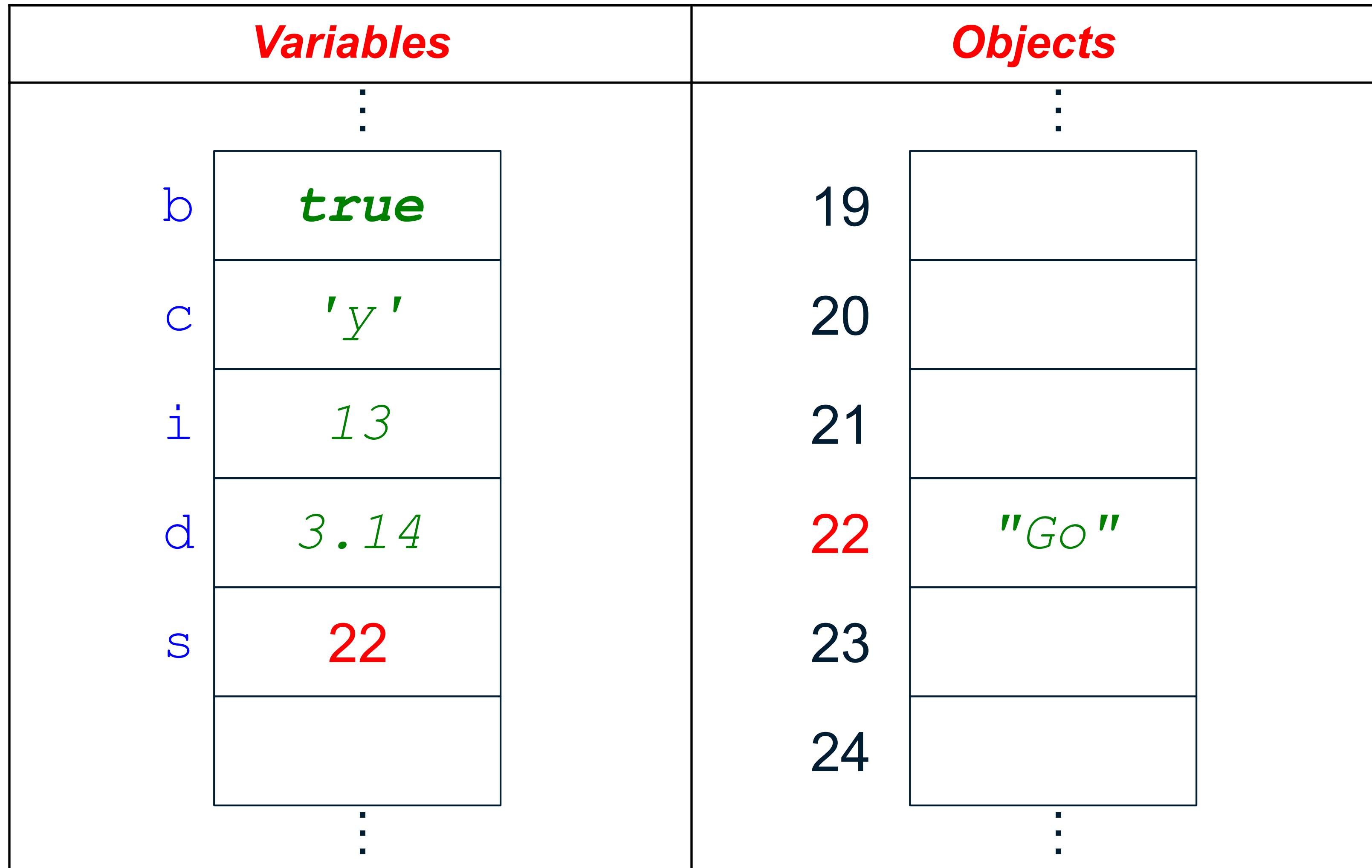
- A reference variable like `s` may be considered to have either of two values:
  - The ***reference value*** of `s` in these pictures is the ***memory address*** at which the object is stored
  - The ***object value*** of `s` in these pictures is the mathematical model value of the object the reference `s` points to, in this case *"Go"*

Reference …

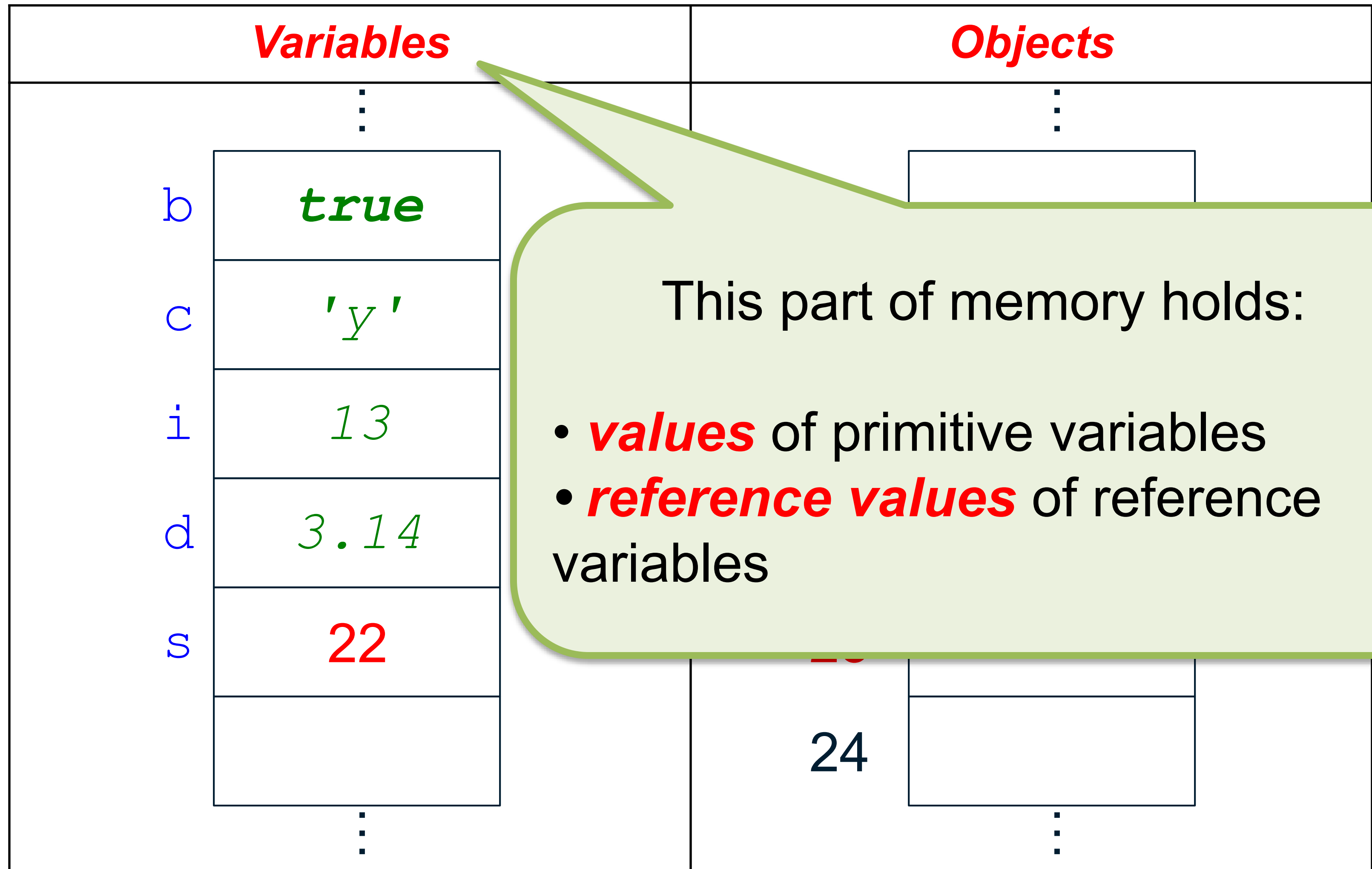
Think of the reference value as simply an “id” or “serial number” of some place in memory.

- A reference vari… considered to hav… …ther of two values:
  - The ***reference value*** of `s` in these pictures is the ***memory address*** at which the object is stored
  - The ***object value*** of `s` in these pictures is the mathematical model value of the object the reference `s` points to, in this case *"Go"*

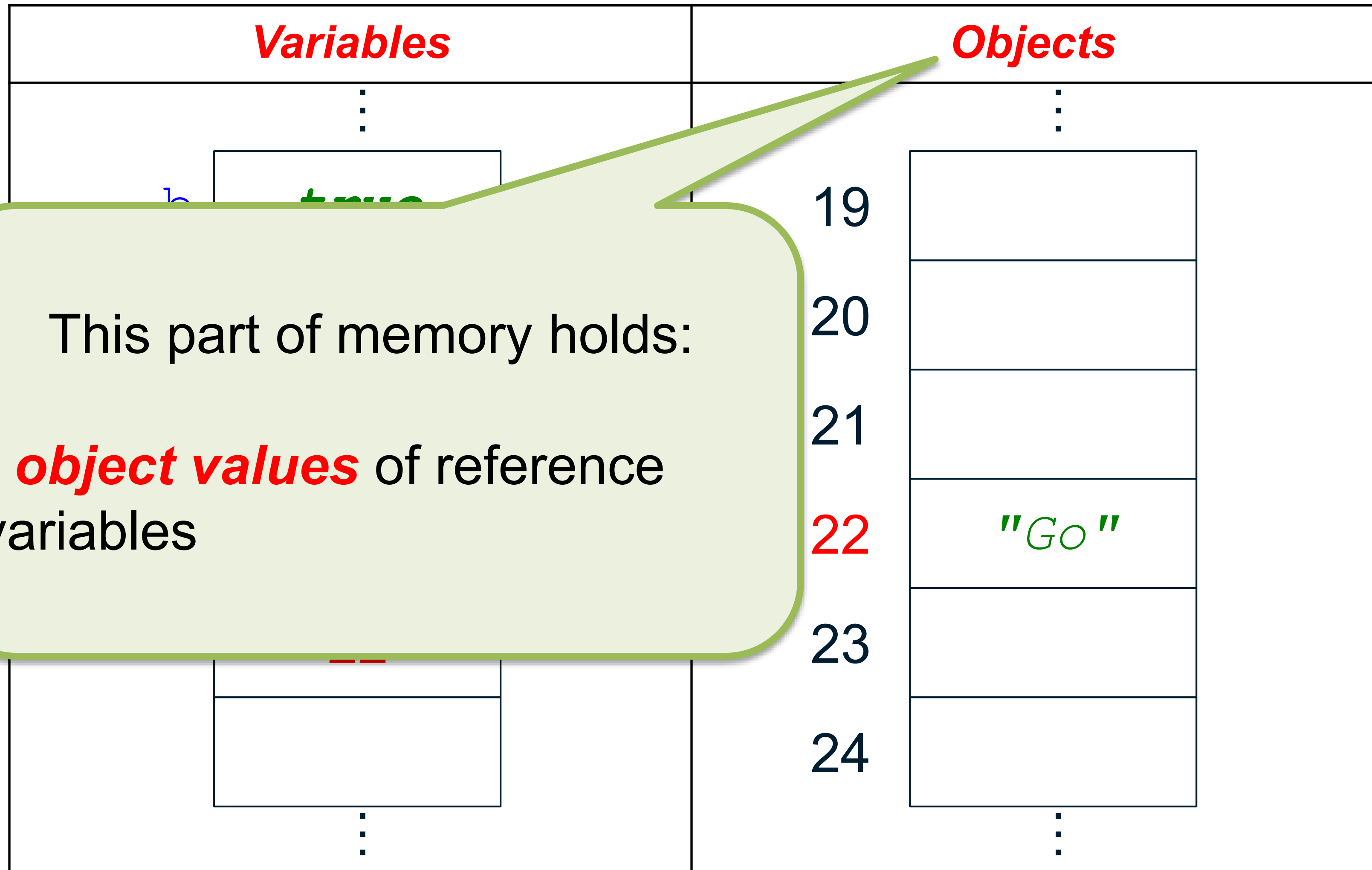
Getting to the "Real Picture"



Getting to the “Real Picture”



Getting to the "Real Picture"



Getting to the "Real Picture"

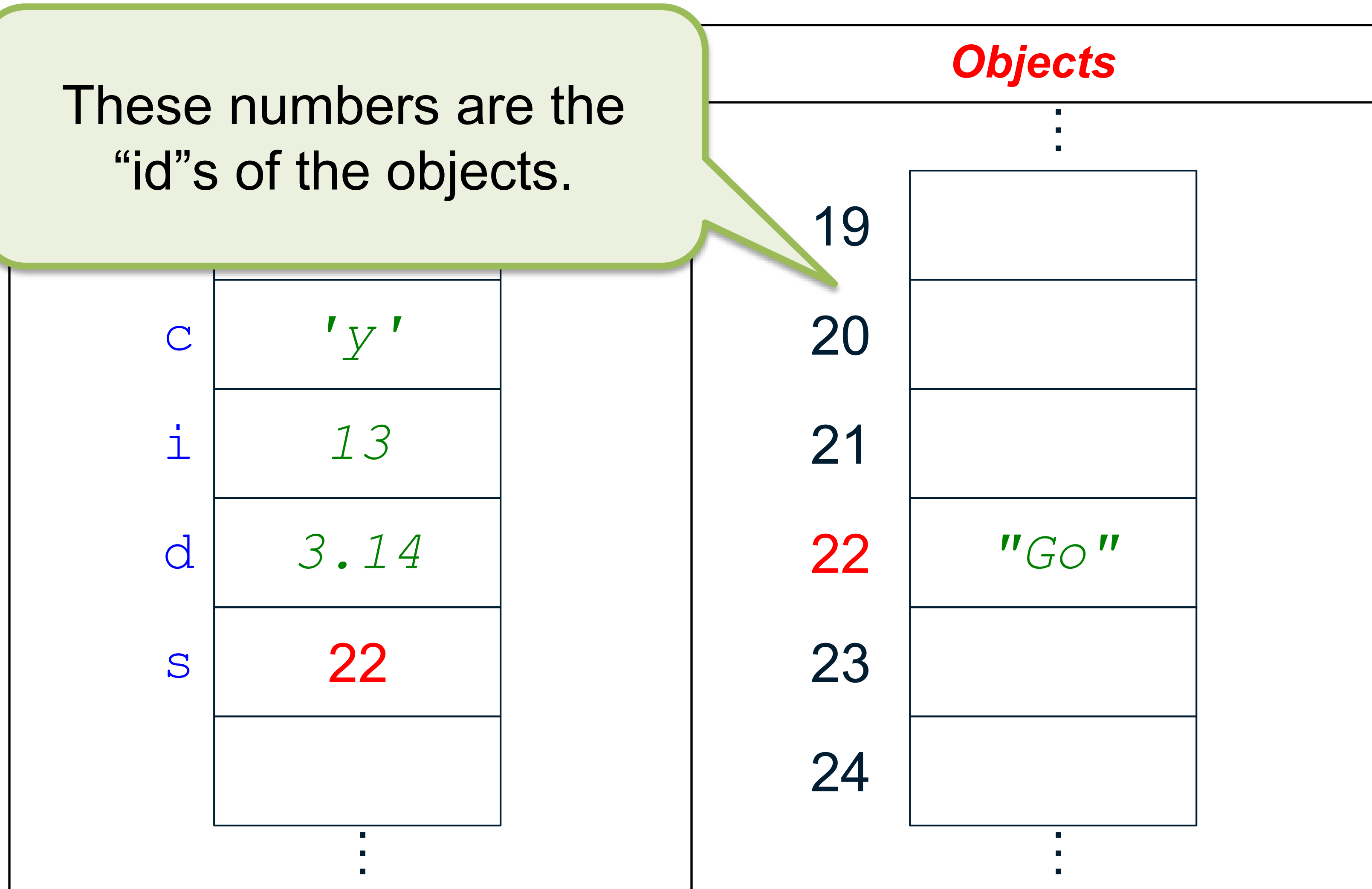
Each object in memory has a unique ***memory address***, or "id"; e.g., this one has id = 22.

c	'y'
i	13
d	3.14
s	22

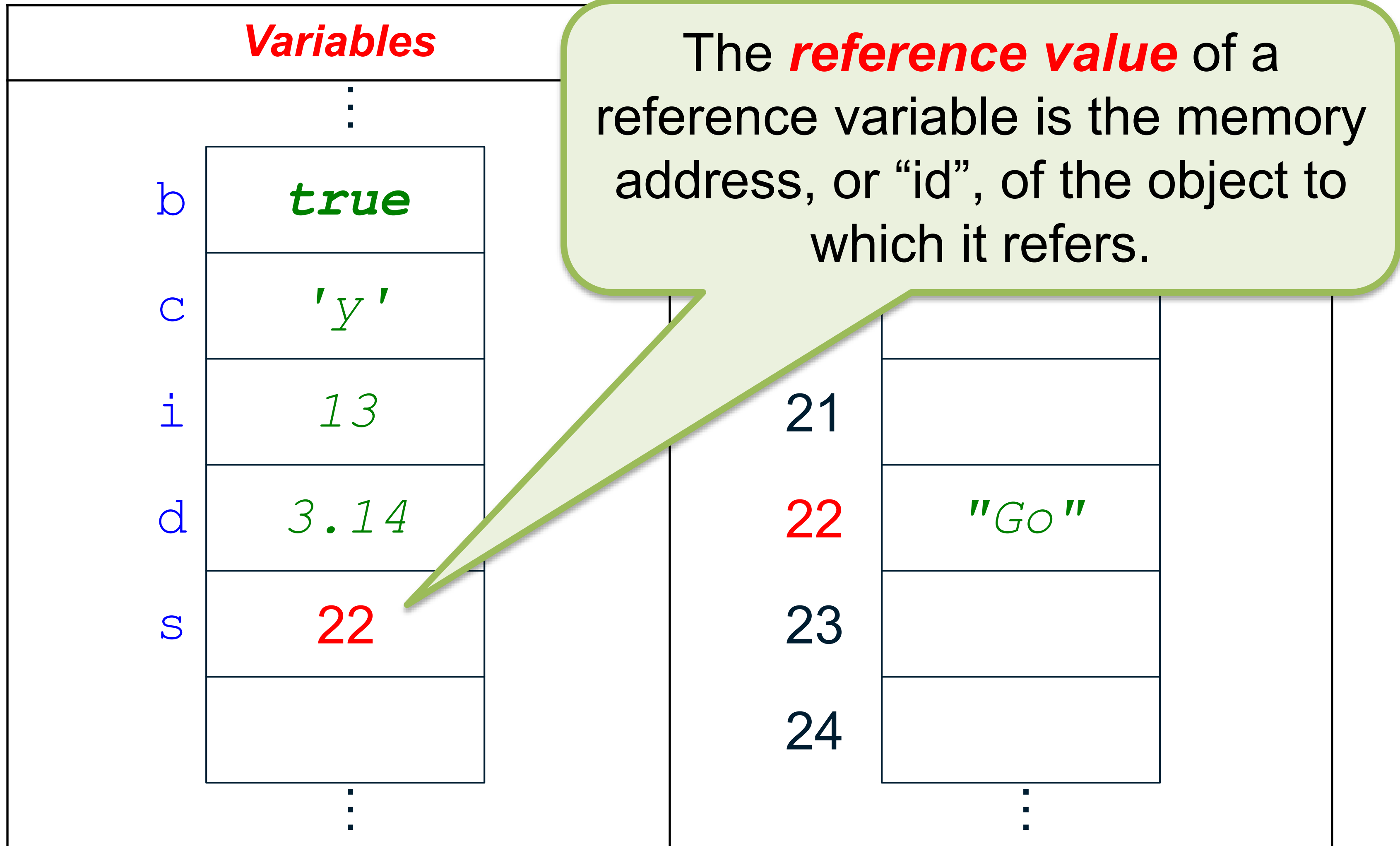
Objects

19	
20	
21	
22	"Go"
23	
24	

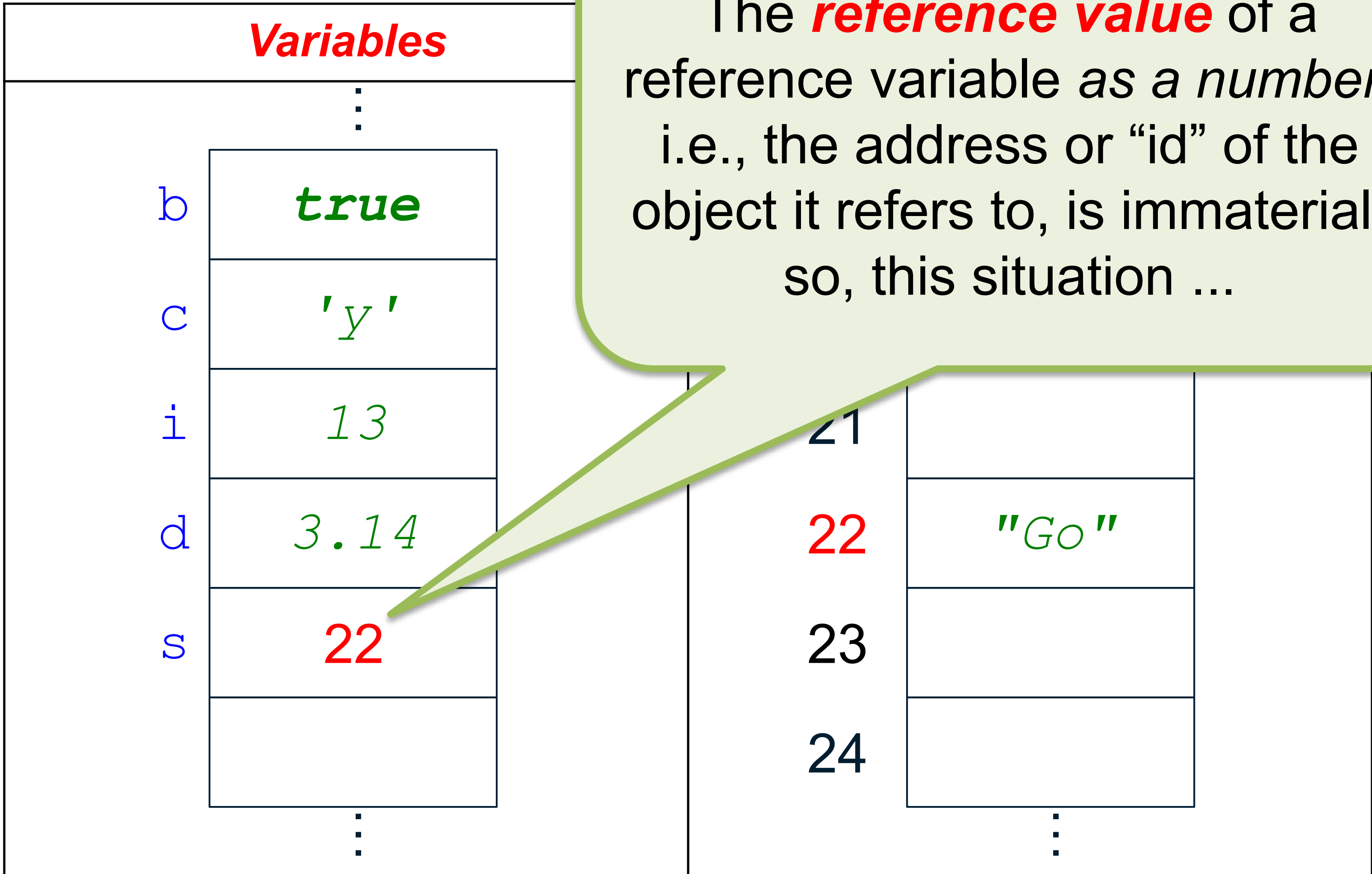
Getting to the “Real Picture”



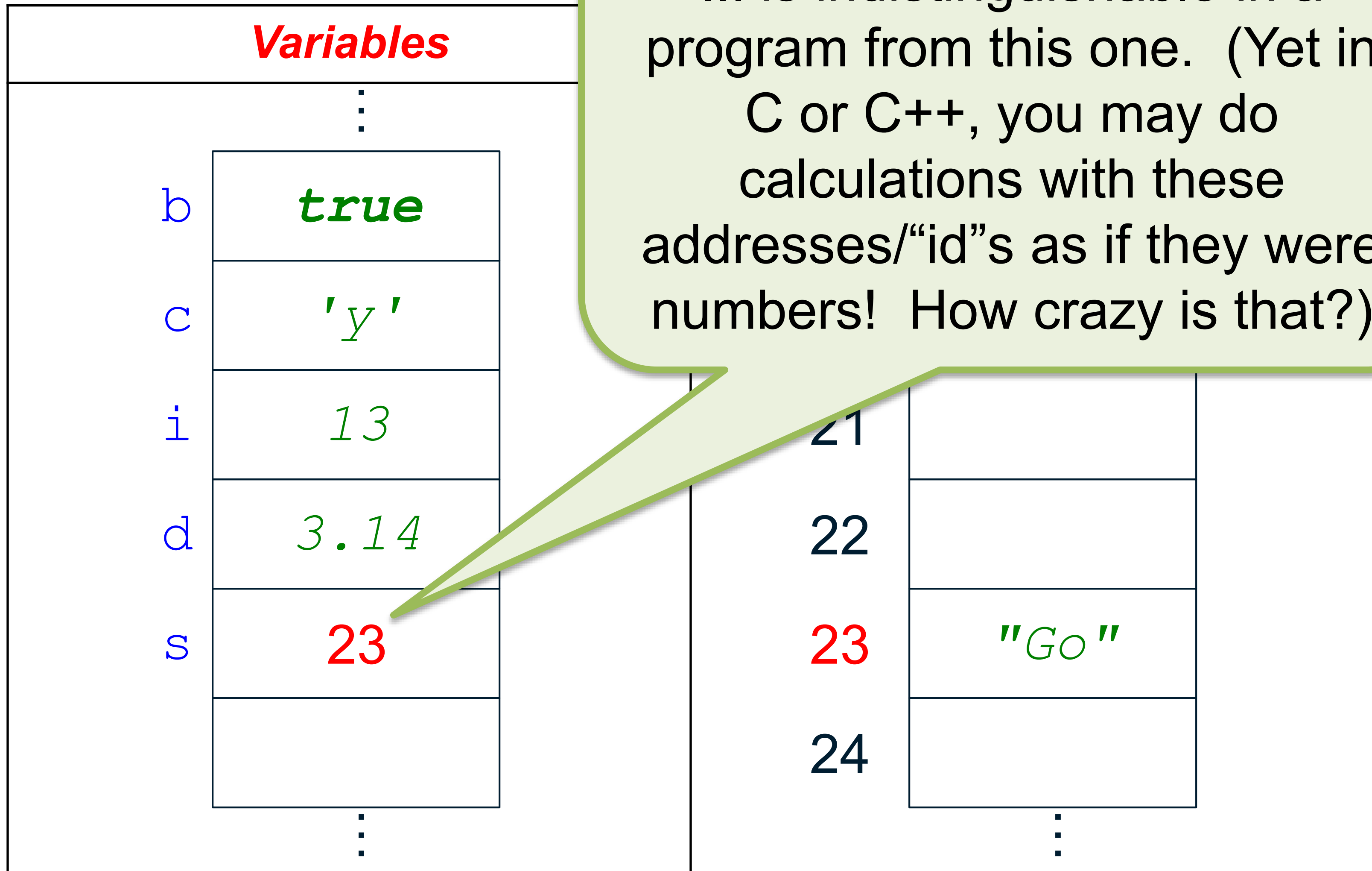
Getting to the "Real Picture"



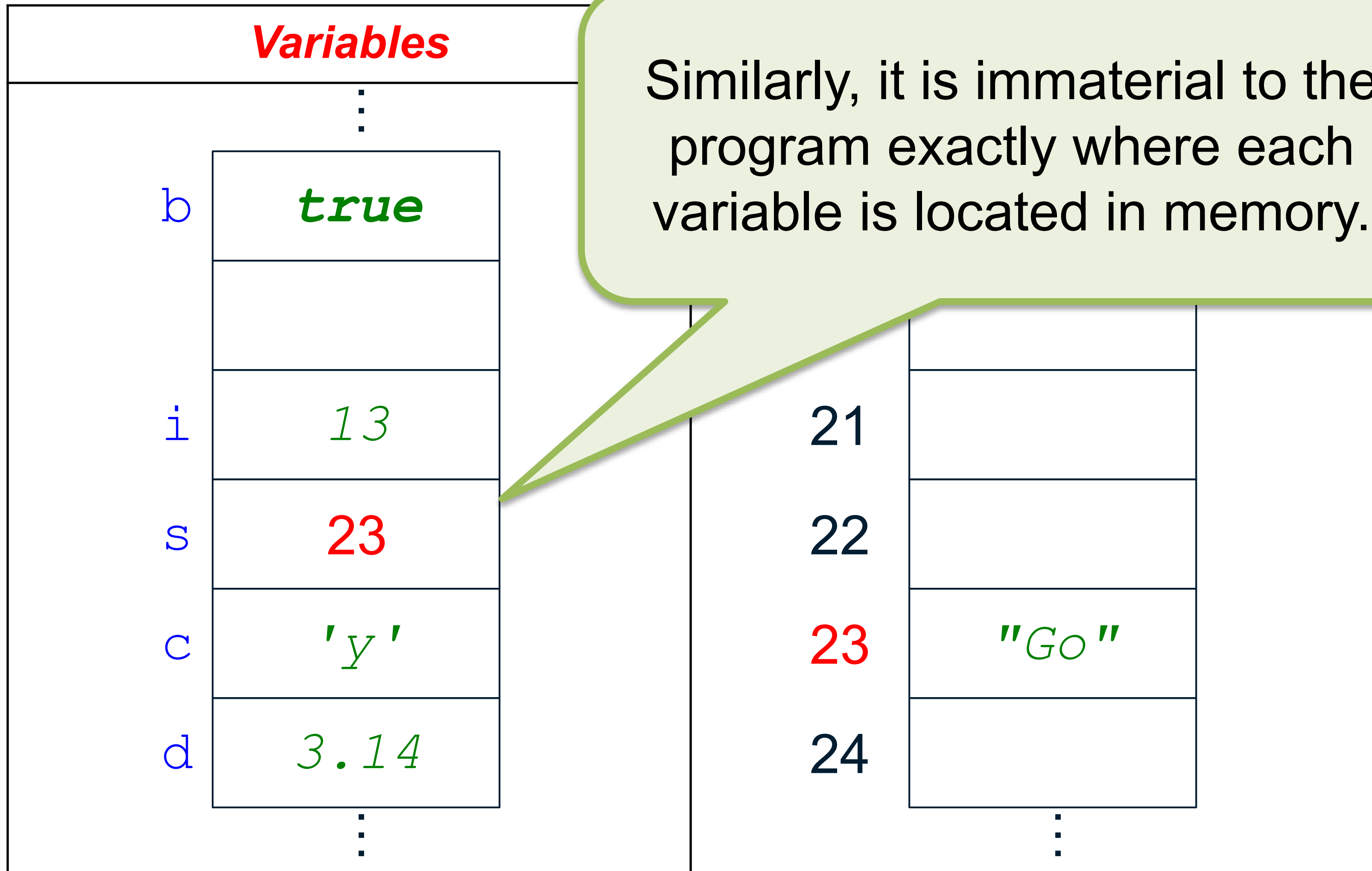
Getting to the "Real Picture"



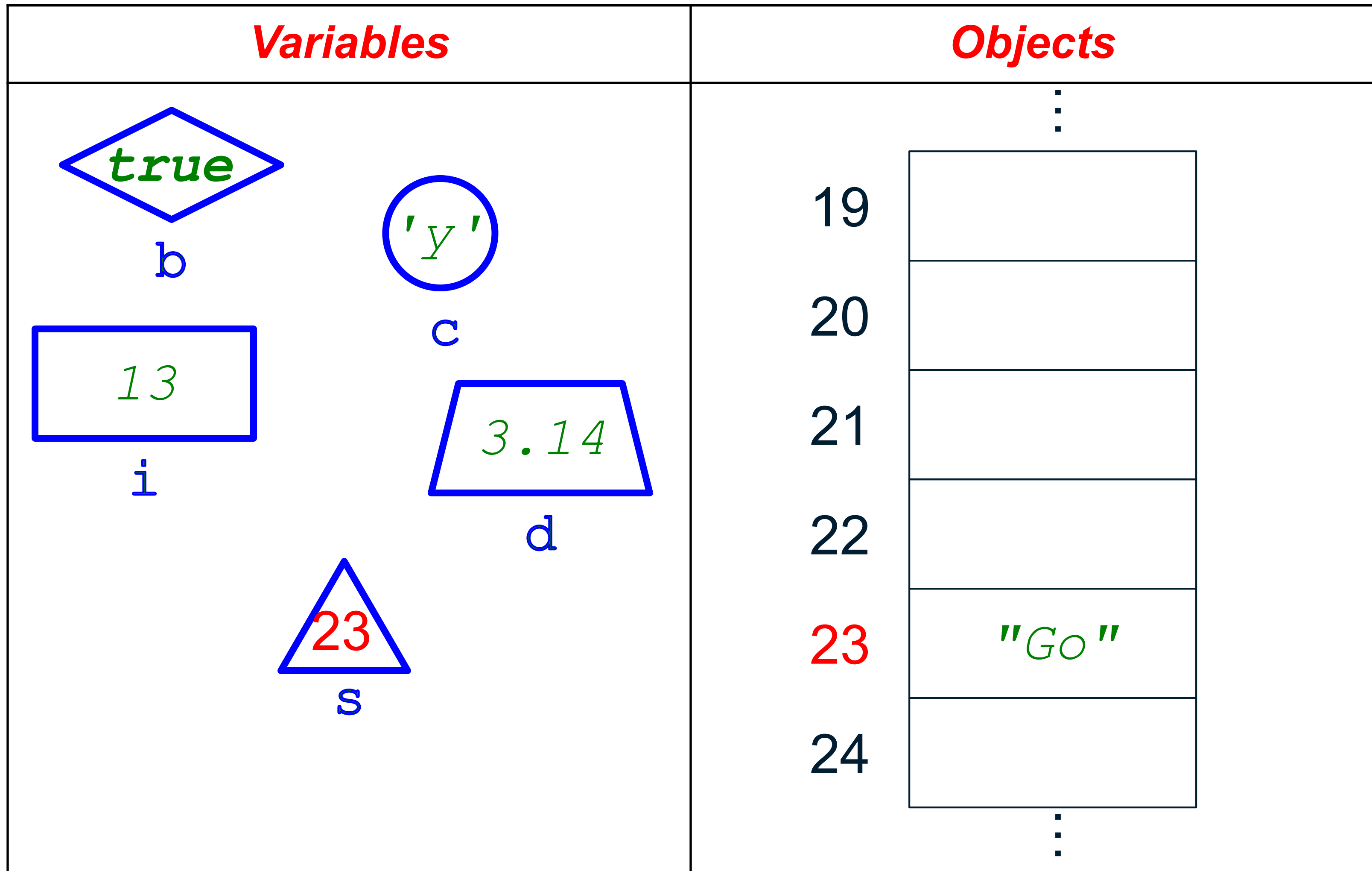
Getting to the "Real Picture"



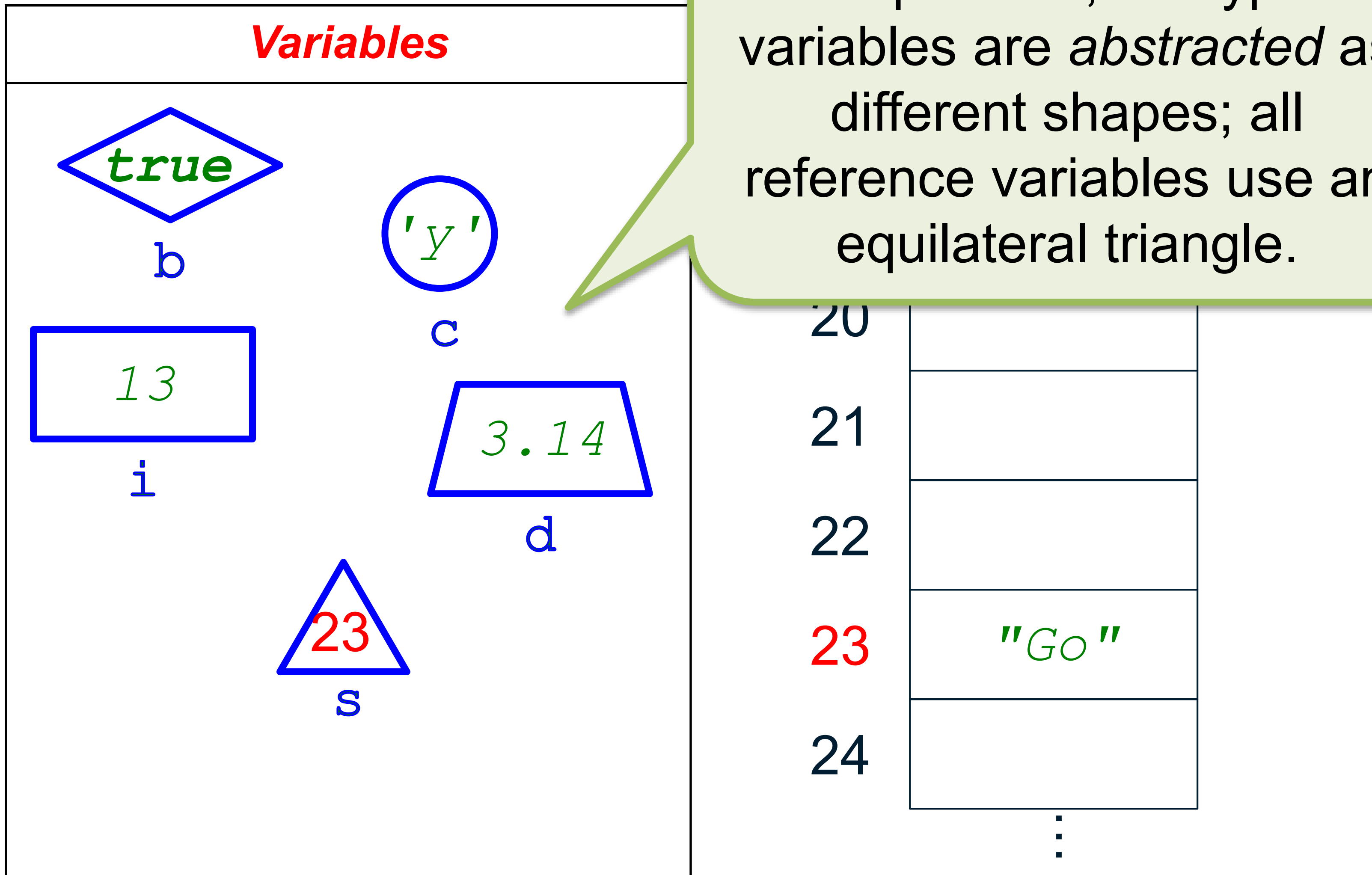
Getting to the "Real Picture"



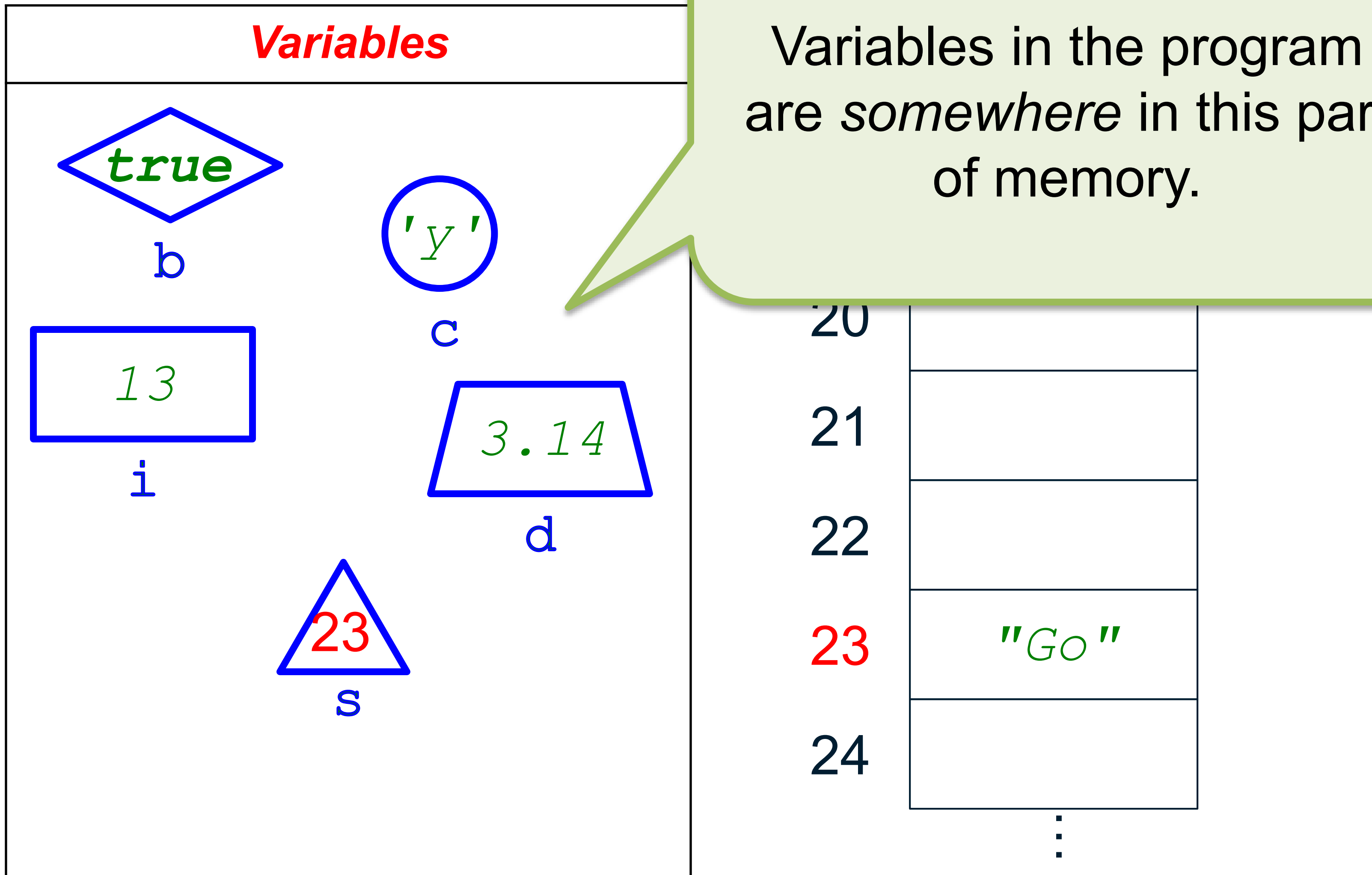
So We Can Simplify...



So We Can Simplify



So We Can Simplify

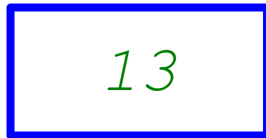


So We Can Simplify...

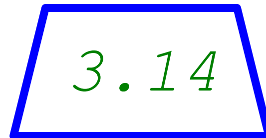
Objects in the program are *somewhere* in this part of memory.

Objects

c



i

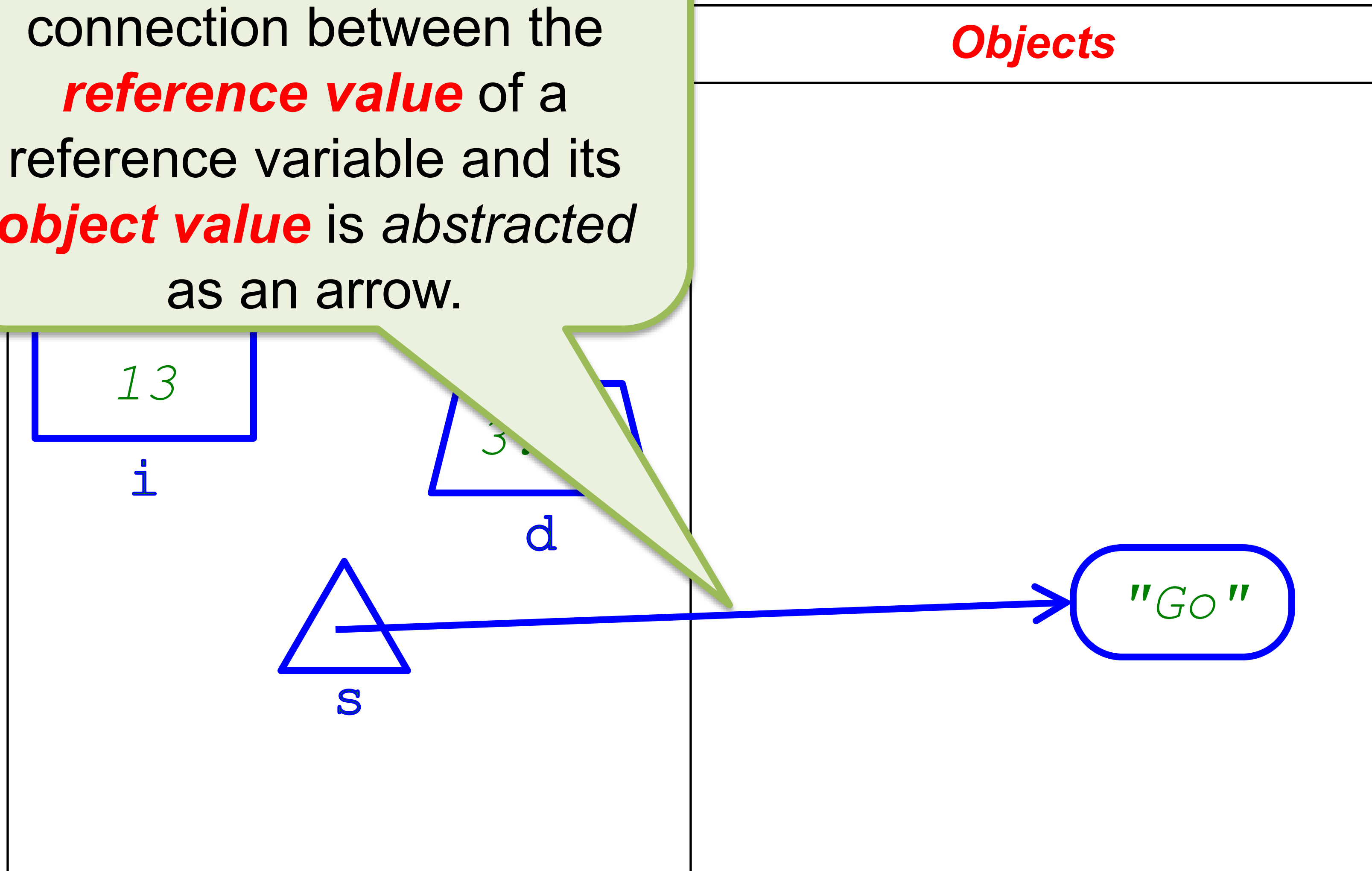


d

s

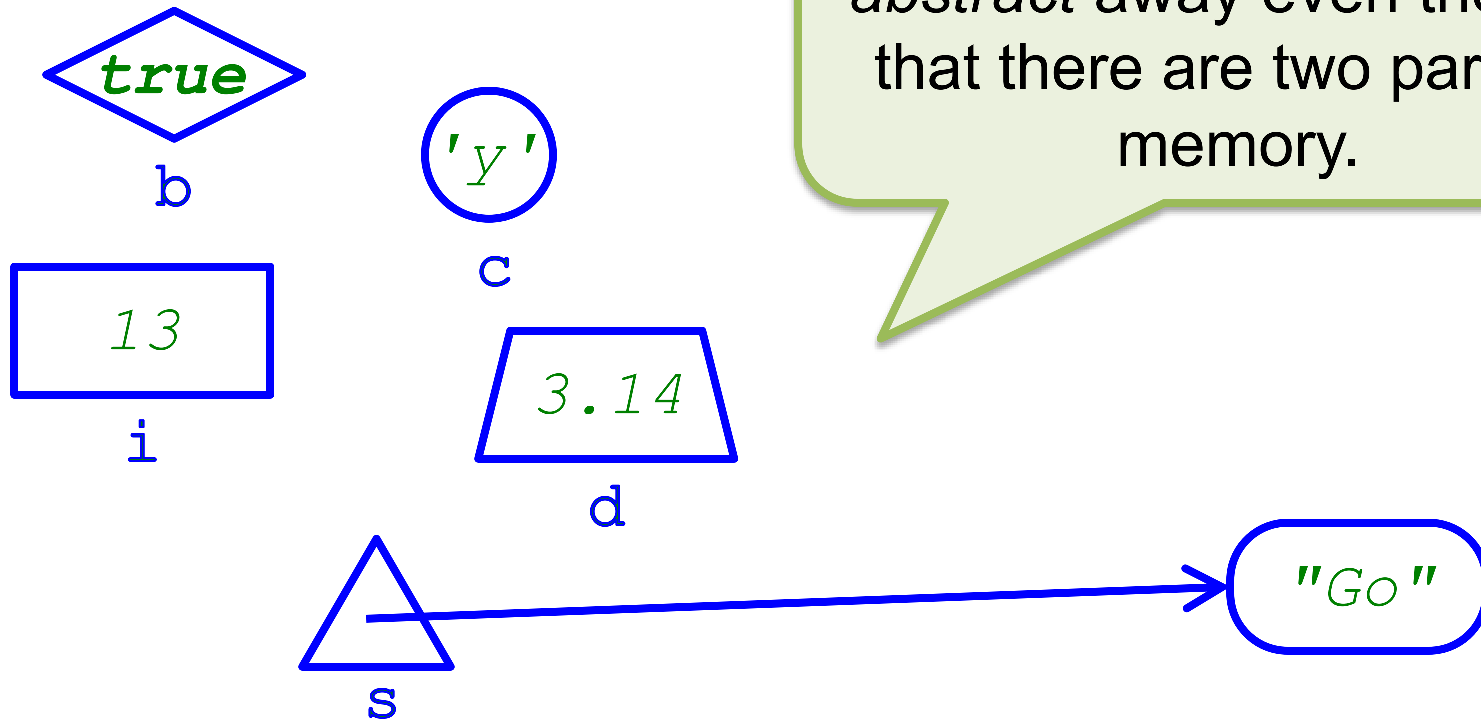
23 "Go"

So We Can Simplify...



Finally...

Our pictures allow us to *abstract* away even the fact that there are two parts of memory.



Notation

- We never care about writing down the ***reference value*** of a reference variable as a *particular numerical* value (though we draw a picture of it: an arrow out of a triangle)
  - So, if you see something like `s = "Go"` in a contract or a tracing table, it *must* mean that the ***object value*** of `s` is the mathematical model value `"Go"`

Notation

- In a tracing table, however, we might want to *remind* ourselves there is a reference involved, so we might record the value of variable `s` using a right arrow instead of an equals sign, e.g., *s* → *"Go"*
  - This means that `s` is a reference variable whose object value is *"Go"*
  - Or: `s` refers to an object with value *"Go"*
  - *Why* would we do this? Coming up...

The Assignment Operator

- The ***assignment operator*** `=` *copies* the value of the expression on the right-hand side into the variable on the left-hand side
- For primitive types, “the value of” can mean only one thing
- For reference types, it could mean “the *reference* value of” or “the *object* value of”
  - Which is it?

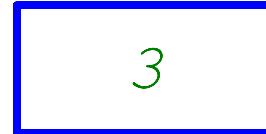
Assignment for Primitive Types

- Consider:

```
int i = k + 7;
```

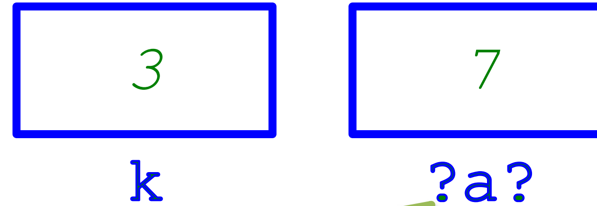
  - First, the expression `k + 7` is evaluated; say *k = 3*, so the expression evaluates to *10*
  - Next, the value *10* is copied into `i`, so after the above statement has finished executing, we have *i = 10*
- How does this happen?

Step by Step: `int i = k + 7;`



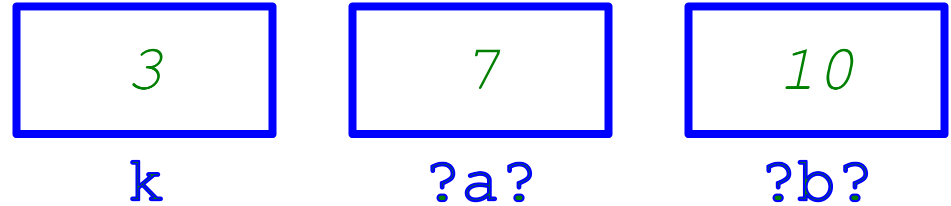
We already have `k`, a ***primitive variable*** whose value is 3.

Step by Step: `int i = k + 7;`



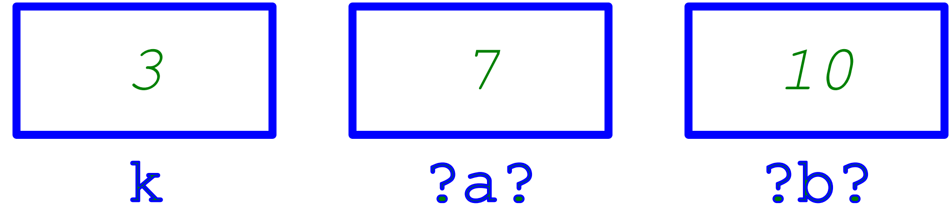
The `int` literal is an ***anonymous primitive variable*** whose value is `7`.

Step by Step: `int i = k + 7;`

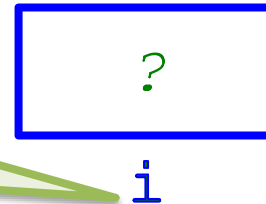


The `int` addition operator `+` results in another anonymous primitive variable whose value is *10*.

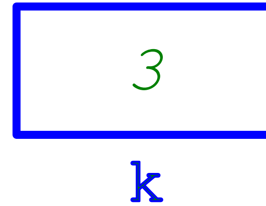
Step by Step: `int i = k + 7;`



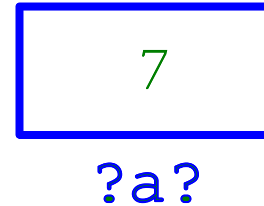
The declaration of the **`int`** variable `i` results in an ***uninitialized primitive variable***.



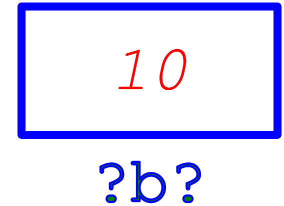
Step by Step: `int i = k + 7;`



k

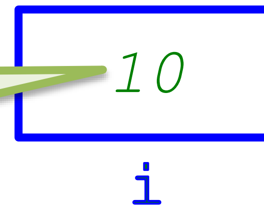


?a?



?b?

The assignment operator copies the value of the right-hand side into `i`.



i

Step by Step: `int i = k + 7;`

3

`k`

The temporary anonymous primitive variables disappear now that the statement has completed executing.

10

`i`

A Tracing Table

Code	*State*
	`k = 3`
`int i = k + 7;`	
	`k = 3` `i = 10`

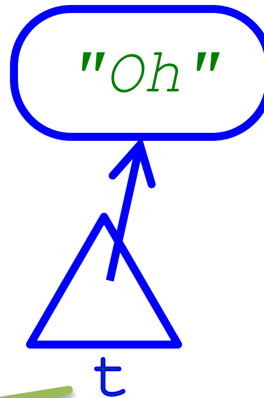
Assignment for Reference Types

- Consider:

```
String s = t + "io";
```

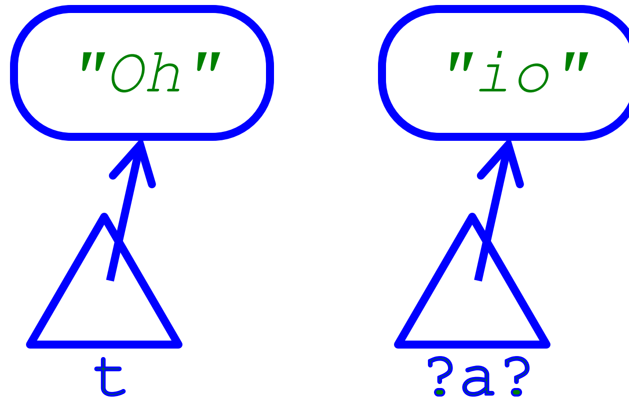
  - First, the expression `t + "io"` is evaluated; say *`t = "Oh"`*, so the expression evaluates to *`"Ohio"`*
  - Next, the value *`"Ohio"`* is copied into `s`, so after the above statement has finished executing, we have *`s = "Ohio"`*
- How does this happen?

Step by Step: `String s = t + "io";`



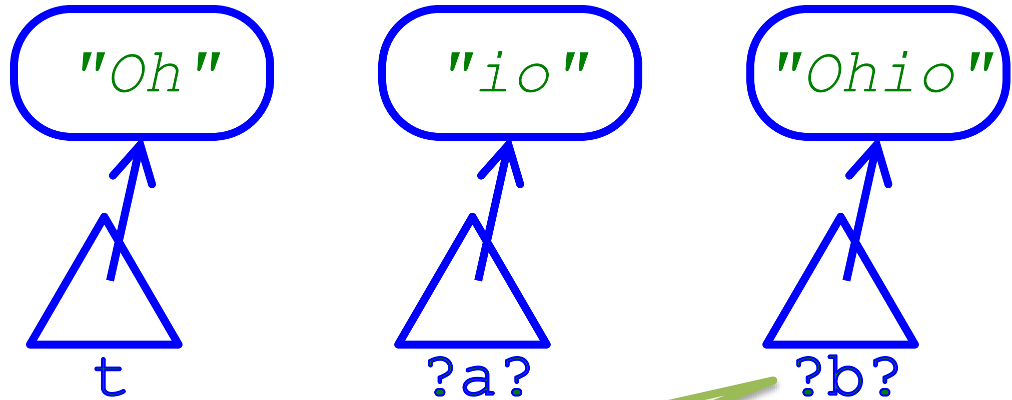
We already have `t`, a ***reference variable*** whose ***object value*** is `"Oh"`.

Step by Step: `String s = t + "io";`



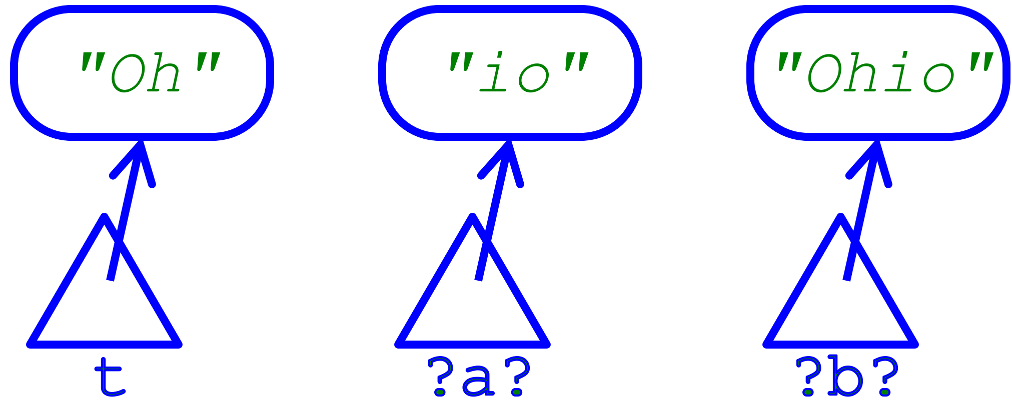
The `String` literal is an ***anonymous reference variable*** whose object value is *"io"*.

Step by Step: `String s = t + "io";`

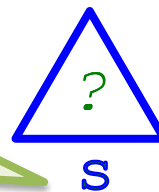


The `String` concatenation operator `+` results in another anonymous reference variable whose object value is `"Ohio"`.

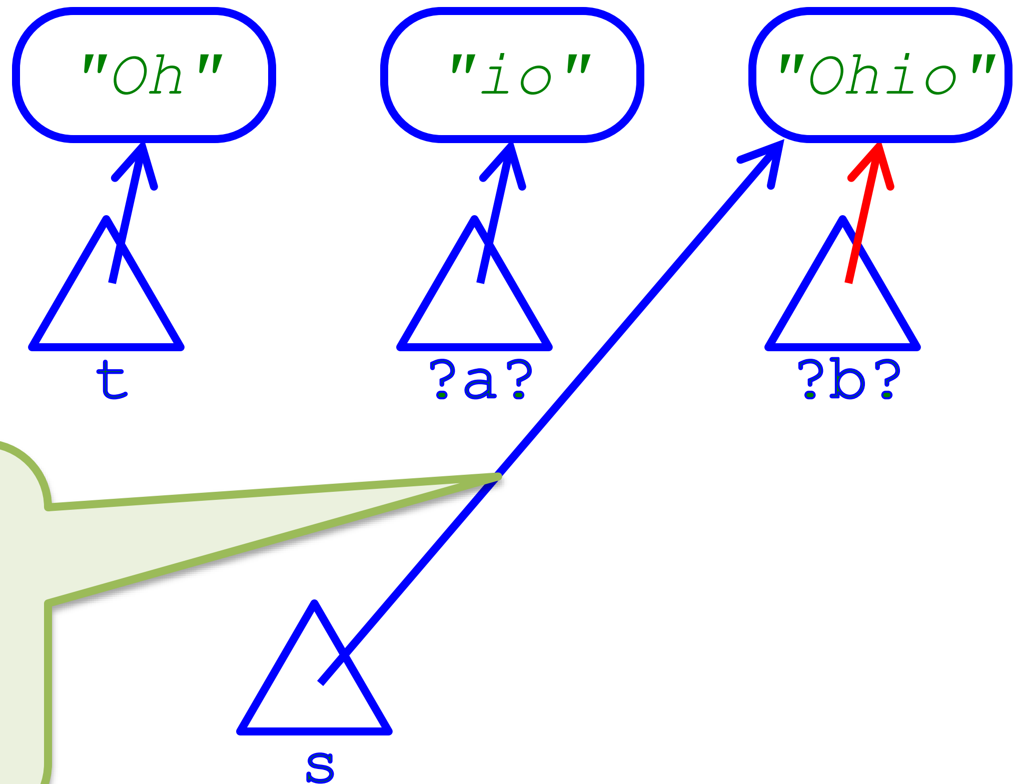
Step by Step: `String s = t + "io";`



The declaration of the `String` variable `s` results in an ***uninitialized reference variable***.

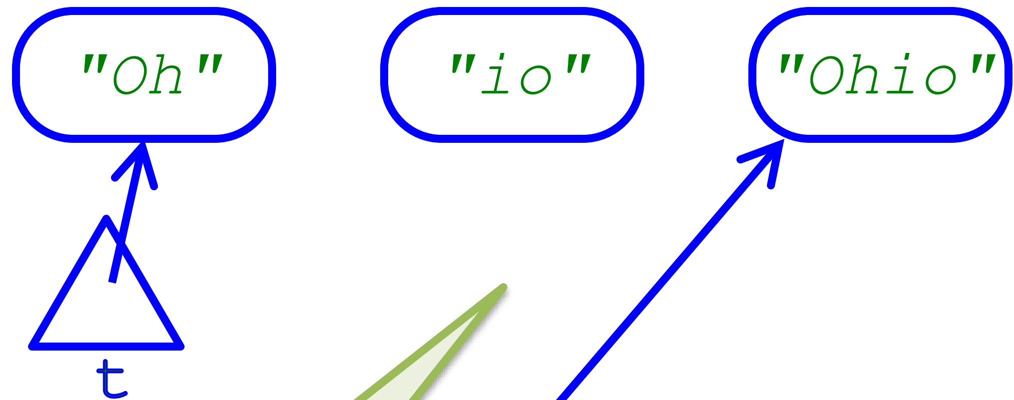


Step by Step: `String s = t + "io";`



The assignment operator copies the ***reference value*** on the right-hand side into `s`.

Step by Step: `String s = t + "io";`



The temporary anonymous reference variables disappear now that the statement has completed executing — but the ***objects*** with values `"io"` and `"Ohio"` remain!

Step by Step: `String s = t + "io";`

"Oh"

"Ohio"

s

Java has a ***garbage collector*** that may come along later and "reclaim" or "recycle" the memory where an unreferenced temporary object is stored; but this does not affect our reasoning.

A Tracing Table Using →

Code	State
	`t → "Oh"`
`String s = t + "io";`	
	`t → "Oh"` `s → "Ohio"`

A Tracing Table Using =

Code	State
	`t = "Oh"`
`String s = t + "io";`	
	`t = "Oh"` `s = "Ohio"`

So What’s Different?

- It seems the net effect of assignment is essentially the same whether we have primitive variables or reference variables
- But not quite...

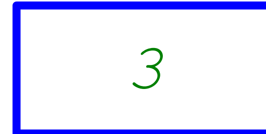
Simplest Assignment: Primitive

- Consider:

```
int i = k;
```

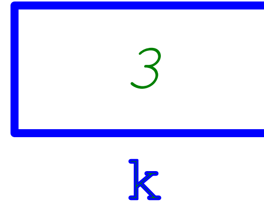
  - First, the expression `k` is evaluated; say *k = 3*, so the expression evaluates to *3*
  - Next, the value *3* is copied into `i`, so after the above statement has finished executing, we have *i = 3*
- Let's do this step-by-step as well...

Step by Step: `int i = k;`

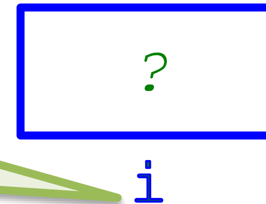


We already have `k`, a ***primitive variable*** whose value is *3*.

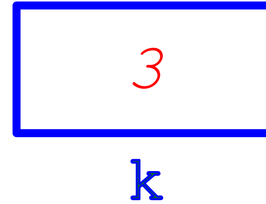
Step by Step: `int i = k;`



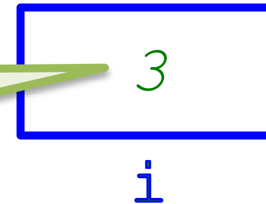
The declaration of the `int` variable `i` results in an ***uninitialized primitive variable***.



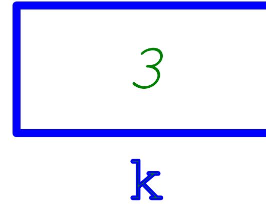
Step by Step: `int i = k;`

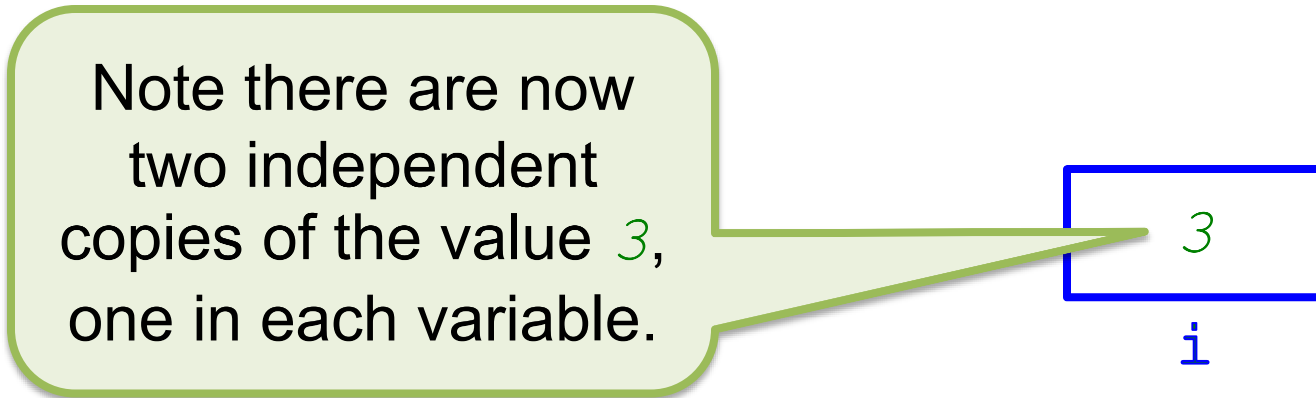


The assignment operator copies the value of the right-hand side into `i`.



Step by Step: `int i = k;`





A Tracing Table

Code	State
	k = *3*
`int i = k;`	
	k = *3* *i* = *3*

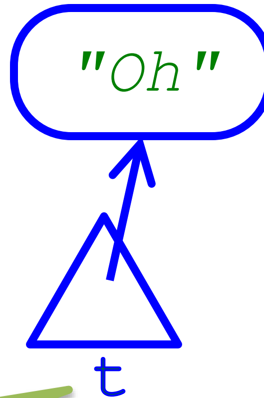
Simplest Assignment: Reference

- Consider:

```
String s = t;
```

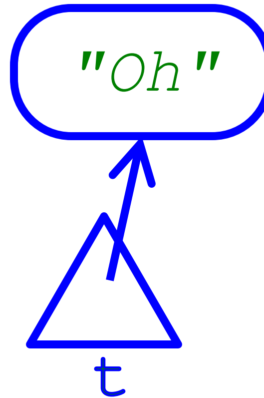
  - First, the expression `t` is evaluated; say *t = "Oh"*, so the expression evaluates to *"Oh"*
  - Next, the value *"Oh"* is copied into `s`, so after the above statement has finished executing, we have *s = "Oh"*
- Let's do this step-by-step as well...

Step by Step: `String s = t;`

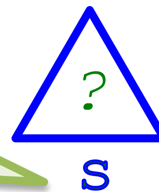


We already have `t`, a ***reference variable*** whose ***object value*** is `"Oh"`.

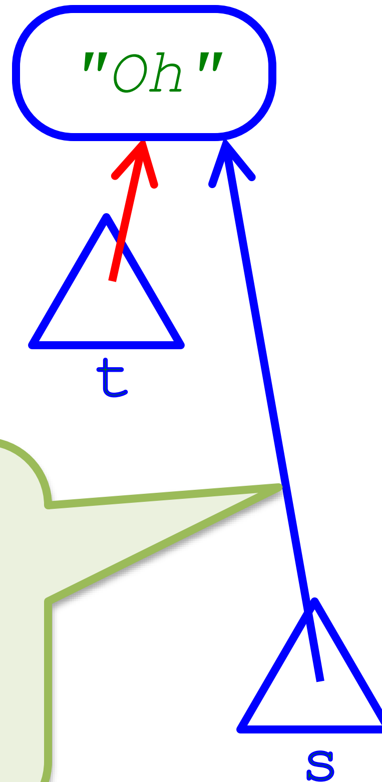
Step by Step: `String s = t;`



The declaration of the `String` variable `s` results in an ***uninitialized reference variable***.

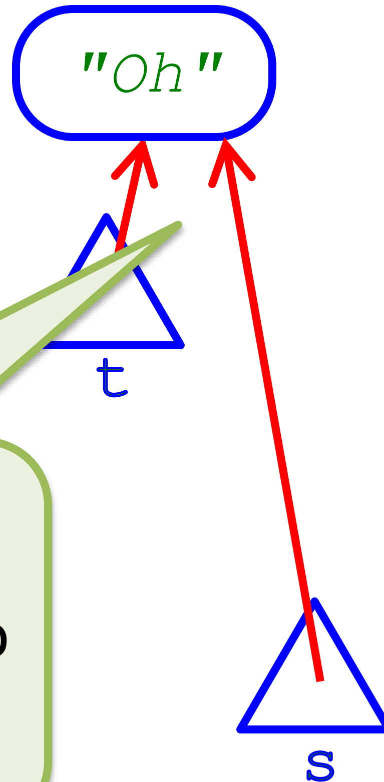


Step by Step: `String s = t;`



The assignment operator copies the ***reference value*** on the right-hand side into `s`.

Step by Step: `String s = t;`



Notice there is still only *one* object but now *two* references to it! These references are called ***aliases***.

A Tracing Table Using ➞

Code	***State***
	t ➞ *"Oh"*
`String s = t;`	
	s,t ➞ *"Oh"*

A Tracing Table Using →

Code	State
	`t → "Oh"`
`String s = t;`	
	`s,t → "Oh"`

The arrow notation helps us remember that there is only *one* object but *two* references to it.

A Tracing Table Using =

Code	State
	`t = "Oh"`
`String s = t;`	
	`t = "Oh"` `s = "Oh"`

A Tracing Table Using =

	State
	t = "Oh"
String s = t;	
	t = "Oh" s = "Oh"

Is this tracing table OK? It suggests there are two separate objects with the value "Oh".

Why It Matters: `NaturalNumber`

Code	State
	z = 99
`NaturalNumber n = z;`	
	z = 99 *n* = 99
`n.increment();`	
	z = 99 *n* = 100

Why It Matters: `NaturalNumber`

But this is *not* what really happens! Try it, step-by-step...

	State
	$z = 99$
`NaturalNumbe` … `= z;`	
	$z = 99$ $n = 99$
`n.increment();`	
	$z = 99$ $n = 100$

Why It Matters: `NaturalNumber`

Code	State
	z → *99*
`NaturalNumber n = z;`	
	z,n → *99*
`n.increment();`	
	z,n → *100*

Why It Matters: `NaturalNumber`

This is what *really* happens!

	State
	z → 99
`NaturalNumbe` `= z;`	
	z,n → 99
`n.increment();`	
	z,n → 100

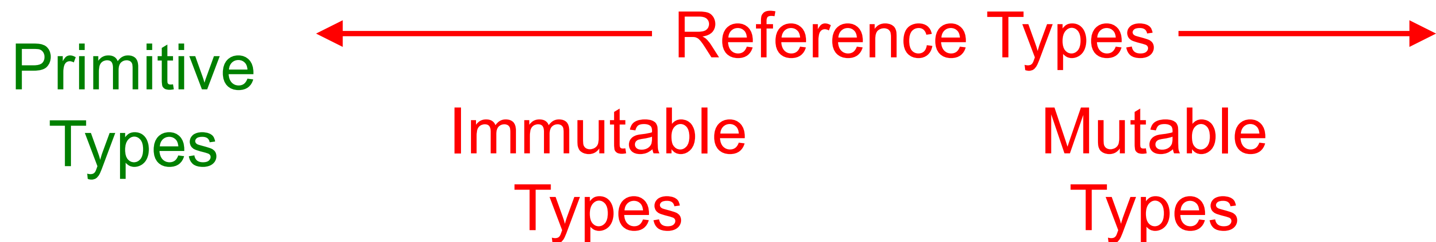
An Important Claim

- The problem illustrated here that arises from ***aliasing of references*** with `NaturalNumber` cannot happen with `String` or `XMLTree`
- What's the difference?

Immutable vs. Mutable Types

- Java reference types are further divided into two different categories:
  - Types for which *no* method might change the value of the receiver, or any other argument of that type, are called ***immutable types***
  - Types for which *at least one* method might change the value of the receiver, or some other argument of that type, are called ***mutable types***

Categories of Types, v. 2

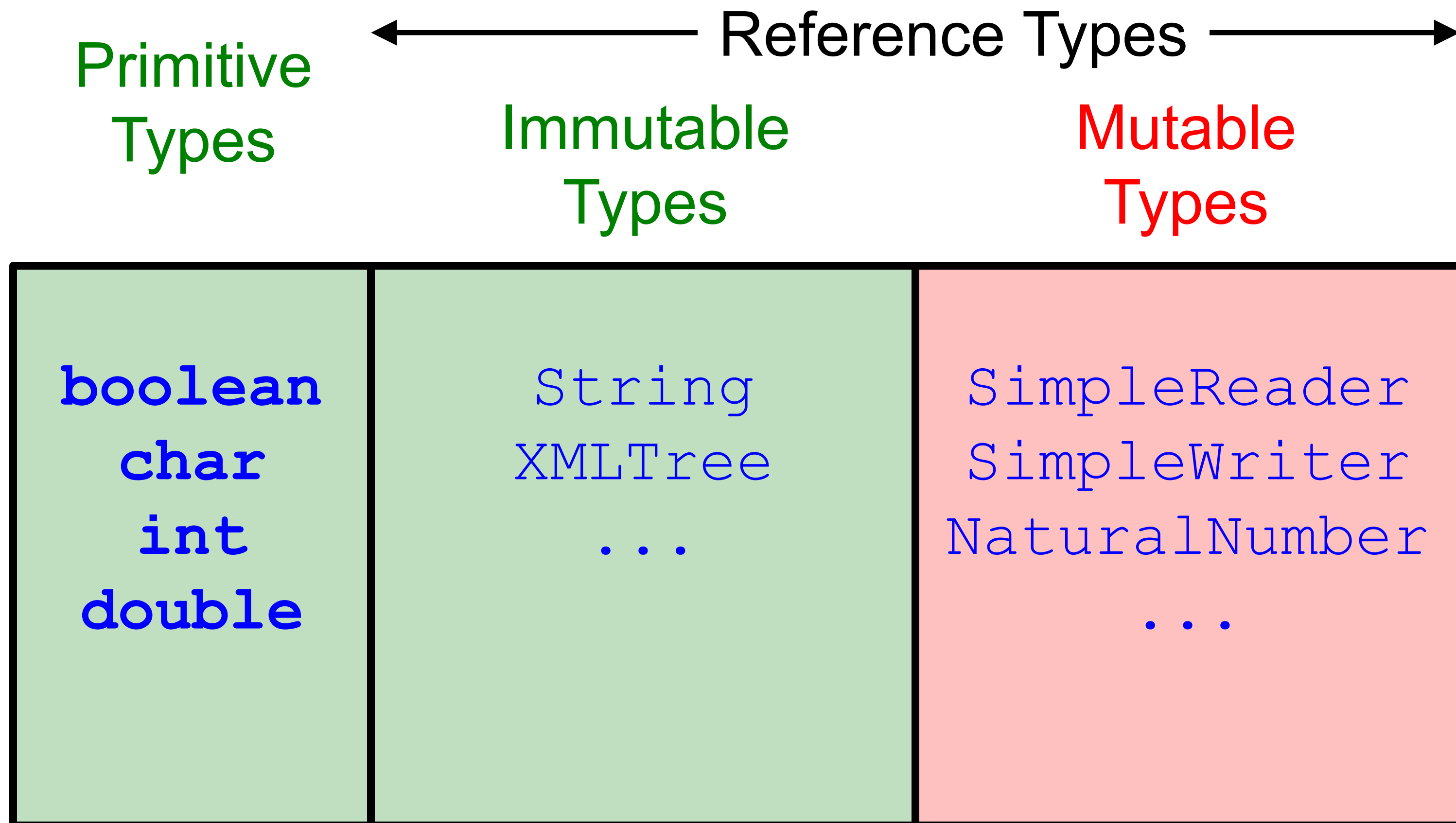


`boolean`	`String`	`SimpleReader`
`char`	`XMLTree`	`SimpleWriter`
`int`	`...`	`NaturalNumber`
`double`		`...`

Restated Claim

- You *may* reason about ***immutable*** types/variables as if they were primitive
- *If and only if there are aliased references*, you *may not* reason about ***mutable*** types/variables as if they were primitive
  - ... because this reasoning short-cut is ***unsound***, i.e., it may predict wrong results compared to executing the code

For Reasoning, It Might As Well Be...



Why Have Mutable Types?

- Couldn't designers of new types just always make them immutable, to simplify reasoning?
  - Yes, but there would be serious efficiency penalties in many cases, so **best practices** dictate that it is more practical to allow mutable types and be especially careful to limit aliasing of references

Parameter Passing for References

- Just as the assignment operator copies reference values, parameter passing to method calls copies ***reference values***
  - The ***reference values*** of the arguments are copied into the formal parameters to initialize them at the time of the call
  - Upon return, nothing is copied back except the returned value of the method (if any), and here too the ***reference value*** is copied back

Complete This Table

Code	State
	m ➞ *143* *k* ➞ *70*
`m.transferFrom(k);`	

Complete This Table

Coc	tate
	$m \rightarrow 143$ $k \rightarrow 70$
`m.transferFrom(k);`	

Consult the contract for `transferFrom`.

Equality Checking for References

- Just as the assignment operator `=` copies reference values, and parameter passing to method calls copies reference values, the equality operator `==` compares ***reference values***

Equality Checking for References

- Since comparing object values is often what you want instead, the `equals` method compares ***object values***
  - At least, **best practices** say it is supposed to
  - Beware: though the `equals` method does what it is supposed to do for nearly all types in the Java libraries (and certainly for all types in the OSU CSE components), in some cases it, too, simply compares reference values!

Example with `NaturalNumber`

Code	State
	$m \to 52$ $k \to 52$
`boolean b =` `    (m == k);`	
	$m \to 52$ $k \to 52$ b = ***false***

Example with `NaturalNumber`

Code	State
	m,*k* → *52*
`boolean b = (m == k);`	
	m,*k* → *52* *b* = ***true***

Example with `NaturalNumber`

Code	State
	$m \rightarrow 52$ $k \rightarrow 52$
`boolean b = m.equals(k);`	
	$m \rightarrow 52$ $k \rightarrow 52$ b = ***true***

Example with `NaturalNumber`

Code	State
	m,k → *52*
`boolean b = m.equals(k);`	
	m,k → *52* *b* = ***true***

Resources

- Wikipedia: Pointer (computer programming)
  - http://en.wikipedia.org/wiki/Pointer_(computer_programming)