# FRAMES AND FOLDERS:

# A TEACHABLE MEMORY MODEL FOR JAVA

Paul Gries, David Gries
Computer Science
Univ. of Toronto, Univ. of Georgia and Cornell Univ.
(416) 978-6322, (706) 583-0395
pgries@cs.toronto.edu, gries@cs.uga.edu

**ABSTRACT**

We present a memory model for use in teaching Java. It includes a notion of class and a way of drawing objects to which students can relate. It includes the frames on the call stack and the steps in executing method calls, including recursive calls. The model starts out simple and is extended as new concepts are introduced, ending up with nested and inner classes.

## 1. INTRODUCTION

We believe that students in the first two programming courses (using Java) should have practice with a model of execution. They should be able to draw instances of classes and subclasses, with enough technical detail to determine the variable or method that is referenced by an identifier within a method body. They should be able to execute method calls by hand, including pushing the frame for a call on the call stack and popping it off when the call is completed. This material is often taught in a later programming language course; we believe it belongs in the first programming course.

Here are just a few examples of the confusions that disappear when students can execute programs by hand. The first concerns the distinction between a reference value and the object to which it refers. If this distinction is not made clear by a model of execution, there is great confusion concerning the assignment statement `v= e;` when `v` and `e` are of class types. The same problem occurs again, more severely, when parameters and arguments are introduced.

Other confusions concern method calls. Students find confusing the idea of pausing a method body while a method call is being executed. Also, consider this code: `f(); g(); f();`. We are often asked what happens to `f` once `g` starts, and we have seen satisfied faces only when we teach our model. Also, to some, the idea that methods are short-lived entities is odd; they want to know why a local variable of `f` doesn't retain its value from one call to the next. The execution model provides the answer.

Students often struggle with scope, trying to determine which variable can be used where, especially with static variables. The confusion is lifted when students can execute a method call by hand, drawing the call stack and using precise rules to determine

the variable to which an identifier refers.

Here's a final reason for wanting to teach a model of execution. We sometimes teach program execution using a debugger, and we want students to use the debugger as well. Explaining parts of the debugger is made easy by our model, since everything seen in the debugger appears in the model. Moreover, if we gloss over how information is maintained in the computer, the students tend not use the debugger.

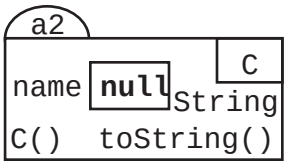
The folder-frame model is used to explain execution of *legal* Java programs. It is not meant to deal with illegal (syntactically incorrect) Java programs, although it can be quite helpful in explaining "why not?" questions.

The frames/folder model first presented here is not in terms of the memory of the computer, and it doesn't use the words *pointer* or *reference*. It describes execution in other, perhaps more accessible (to the student) terms. But one of the authors has successfully taught (and prefers) the same execution model in terms of computer memory, using words like *address* and *pointer*. Therefore, in the last section of the paper, we present the same model, but directly in terms of the computer. Our instructions for executing a program are the same for both models. The reader can adopt either one.

Sect. 2 outlines the basic idea for dealing with classes and objects. Sect. 3 shows how to execute method calls. Sect. 4 discusses a suitable order for introducing topics in an introductory programming course. Sect. 5 shows how easily inner classes fit into the model (although they are rarely taught in a first course). Sect. 6 introduces the same model of execution but in terms of the computer. Space restrictions force us to be incomplete and terse; a complete description can be found on our websites, www.cs.uga.edu/~gries and www.cs.toronto.edu/~pgries. We use this model in our livetext, *ProgramLive* [1], and its paperback *Companion to ProgramLive* [2].

## 2. CLASSES AND OBJECTS


We present classes and objects in terms of something students already understand. We view a class `C` as a drawer of a file cabinet. The drawer contains two kinds of information. It contains all the static components defined in the class —the static variables and methods, perhaps each on its own sheet of paper. A reference like `C.MAX_VALUE` is evaluated by looking in `C`'s drawer, picking out static variable `MAX_VALUE`, and using its value. We sometimes call it the *class box*, instead of the class file drawer.

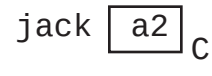



The drawer also contains all instances (objects) of class `C`, each drawn as a manila folder. The name, or label, on the tab of the manila folder identifies the folder. Whoever creates a new folder (object) chooses a new name for it. When we create a folder by hand, we choose the name; when the computer creates a folder during execution, it chooses the name. This name can be abstract, like `a2`, or it can look like a memory address, say in hexadecimal, as the instructor prefers. Java hides memory addresses, so the name format is unimportant. But the name must be unique.

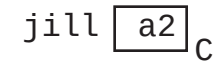
The name of the class is written in a box in the upper right of the folder, so it is always clear where the folder came from —which drawer it belongs in.

The folder contains all the non-static members given in the class definition. Each instance variable appears as a named box, with the value of the variable within the box. When it is useful, we put the type of the variable at the lower right of the box. For each instance method, we write its name in the folder, along with its parameters.

Students are often confused as to just what a class is. Does it get executed? If not, what does it do? As an experiment, ask your students what a class is, and see the confused answers that you get. With our model; the confusion goes away. A class is simply a file drawer, and the class definition defines what goes in it.

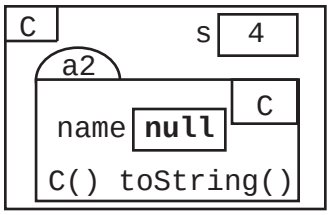



For a variable `jack` of class-type `C`, execution of the assignment `jack= new C();` creates a new folder (e.g. the one above) and stores in `jack` the name that appears on the tab of the folder. During execution, variable `jack` is said to contain the name of that folder.




The name of the folder that is in variable `jack` in the above paragraph is just another value. So, when an assignment `jill= jack;` is executed (where variable `jill` is also of type `C`), the name in `jack` is simply copied to variable `jill`.

Aliasing is easy to understand in this model. In an office, suppose someone gets a manila folder from a file drawer, changes something in the folder, and puts the folder back. Later, anyone looking in the folder will see the change. In our running example, variables `jack` and `jill` contain the same manila-folder name, `a2`. If jack changes variable `name` using, say, `jack.name= "abc";`, then, later, when Jill evaluates expression `jill.name`, the value retrieved will be the value that `jack` stored in `name`.




The model allows us to explain references to static components. Suppose class `C` has a static variable `s`. (`C`'s file drawer is pictured to the left, with its name in the upper left). Then, anything within `C`'s drawer has direct access to `s`, through that name: `s`. But a method that is in another file drawer (in another class) must use `C.s` —obviously, the method must indicate what file drawer `s` is in. Here, we are using the *inside-out rule*:

> **Inside-out rule.** Suppose a folder or drawer, `Inner`, is inside another folder or drawer, `Outer`. Then, the methods that are in `Inner` can reference each component `x` of `Outer` directly (unless `Inner` also defines `x`).

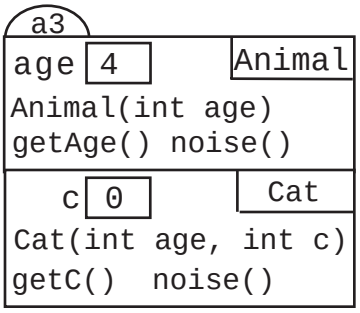
Later, this inside-out rule is included in our rules for finding the target of a reference, using a lot more technical detail and the term "enclosing scope". But the term *inside-out rule* is a nice handle for the students to grasp.

## 2.1 Folders (Instances) of Subclasses

Consider the following two classes (we omit most specifications of classes and methods since they are not relevent to this discussion):

```
public class Animal {
    private int age;
    public Animal(int age)
        { this.age= age; }
    public int getAge()
        { return age; }
    // = noise this Animal makes (the empty String)
    public String noise()
        { return “”; }
}
```

```
public class Cat extends Animal {
    private int c;
    public Cat(int age, int c)
        { super(age); this.c= c; }
    public Cat getC()
        { return c; }
    // = noise this Cat makes
    public String noise()
        { return ''meow''; }
}
```




The manila folder for an instance of subclass `Cat` of class `Animal` is shown to the left. The components defined in the superclass appear in the upper partition; as before, the name of the class (`Animal`) appears in a box in the upper right. The components defined in the subclass appear in the lower partition; the name of the subclass (`Cat`) appears in its own box in the upper right. Both parts of the object appear as a single unit, in one manila folder.

The format of a folder is extended to subclasses of subclasses in the obvious way. For example, suppose class `Siamese` extends `Cat`. An instance of class `Siamese` is drawn with three partitions: components of `Animal` in the top one, components of `Cat` in the middle one, and components of `Siamese` in the bottom one.

In principle, one should draw a partition for class `Object` at the top of every folder. After discussing this issue with students, we explain that we omit it simply to keep the drawing of folders simple, but it really is there.

Later, we will explain how the folder-view of an instance helps in determining which method is used when a method name is used in a call.

## 2.2 Casting: Real and Apparent Views of Objects

An expression `e` that yields the name of a folder has a class-type. This concept is crucial to explaining inheritance, shadowing, and casting. The *apparent* view of expression `e` is the type that Java determines for `e` from a syntactic analysis (at compile-time). The *real* view of `e` is the type of the folder that `e` names, which is always the class given in the bottom partition of a folder. At compile time, only the *apparent* view is used to determine what is legal.

Consider classes `Animal` and `Cat`, defined above, and these statements:

```
Cat c= new Cat(5,4);  Animal a= c;
```

After execution, `c`'s apparent view is `Cat`, and its real view is also `Cat`, because `c` contains an instance of `Cat`. Variable `a`'s apparent view is `Animal`, but its real view is `Cat`. Because `a`'s apparent view is `Animal`, expression `a.getC()` is illegal. Only these expressions are legal: `a.age()`, `a.getAge()`, and `a.noise()`.

Some students have difficulty grasping the above until they see the reasons for it. Our model, together with this example and the introduction of the terms *real* view and *apparent* view, clears up the confusion. From the apparent view, the object is just an `Animal`, and `Animals` don't have a method `getC`.

When first asked, students will say that the method referenced by `a.noise()` is the one in the apparent view, `Animal`. We then say that OOP wouldn't be half as useful as it is if this were the case. Since this `Animal` is really a `Cat`, method `noise` in `Cat` should be referenced. In summary: the apparent view is used to determine syntactic legality, but at runtime, the real view is used in determining the method to call.

The following rule, which is incorporated in the rules given in Sect. 3.2, is useful. There is an extremely special case where it doesn't work: when a private method of a superclass is redeclared in a subclass and v is of the superclass type.

> **Overriding rule** Consider a reference `v.x(...)` to a method in the folder named by `v`. To determine which method it refers to, search the folder for the method in an upward direction, starting at the bottom of the folder named by `v`.

## 3. EXECUTING METHOD CALLS

We believe that one firmly grasps the idea of methods and method calls only when one can execute a method call by hand, drawing the frame for the call and all the variables and objects created during execution. We now present a model of method-call execution.

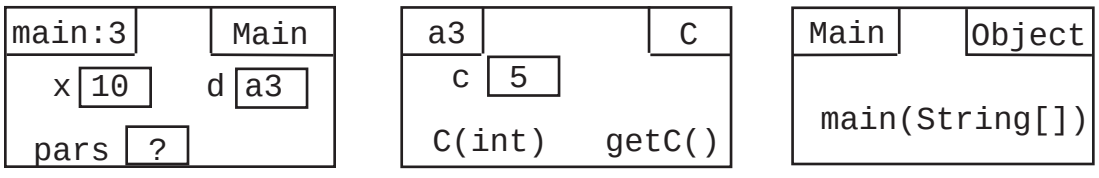
When a method is called, a *frame* is created. The frame is simply a box that holds information that is used during execution of the method body. When the method call is completed, the frame is erased. Frames for uncompleted method calls are kept on the *call stack*. The top frame on the call stack is called the *active* frame. Most debuggers show the call stack during execution.

We now describe the content of a frame, using as an example the method calls in the following two classes. Of course, when first introduced, frames have less information, and they become more complex as new concepts are introduced. See Sect. 4.

```
public class Main {
    public static void main
            (String[] pars) {
        int x= 10;
        C d= new C(x-6);
        x= d.getC();
    }
}
```

```
public class C {
private int c;
public C(int pc)
    { c= pc; }
public int getC()
    { return c; }
}
```

Below, we show the frames for the call on `main` and the call `d.getC()` and also instance `a7` of class `C`, assuming that method `getC` of folder `a7` was called.




a frame | an object | a class

Each frame contains the parameters and local variables of the method, written as variables (named boxes). In addition, a frame contains two subboxes.

The subbox in the upper left corner of a frame contains the name of the method being called and a line counter (or program counter), which is the number of the line in the

method body that contains the current statement being executed (we assume each line has at most one statement). If the line is complex, for example, `f() + g(h(x));` then the line counter must indicate (possibly using a roman numeral) which call is currently being executed.

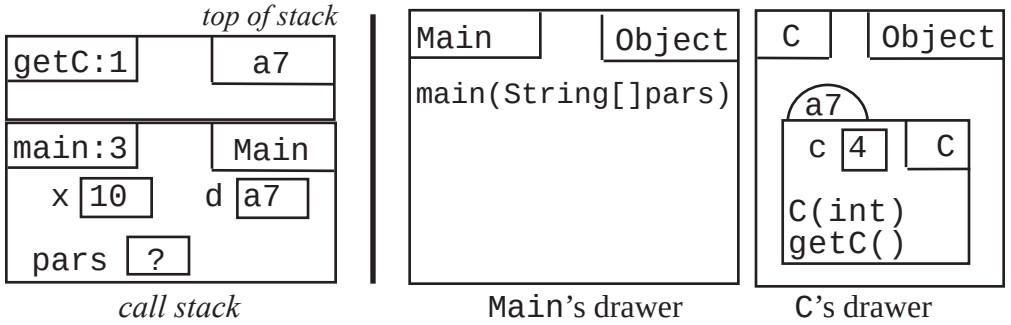
The subbox in the upper right of the frame is called the *scope box*. The scope box is used to answer the question, "Where do we look next for the variable or method (if it is not in the frame)?". We define the contents of the scope box:

1. For a static method, the scope box contains the name of the class in which the method is defined.
2. For a nonstatic method (a constructor is nonstatic), the scope box contains the name of the object in which the method appears.

We now define exactly how to execute a method call (by hand). Note that the portion of the call stack just above active frame is used as communication between the calling and called programs; argument values for a new call are placed there, and return values are placed there when a function body terminates.

1. Evaluate the arguments of the call, from left to right, and push their values onto the call stack.
2. Draw the rest of the frame on the call stack, filling in the subboxes appropriately.
3. Label each argument on the call stack with the corresponding parameter name.
4. Draw the local variables in the frame, as named boxes.
5. Execute the method body, one statement at a time. After executing a statement, change the line counter (in the upper-left subbox of the frame) accordingly. Use the rules given in Sect. 3.2 for finding the targets of identifiers within the method body.
6. Upon termination of the method body: For a function, i.e. a method that returns a value, pop the frame from the call stack and push the value to be returned onto the call stack; for a procedure, just pop the frame from the call stack.
7. Continue executing at the place designated by the line counter in the method now on top of the stack.

So that students don't get confused, we ask them to draw the call stack on the left and file drawers and folders on the right, with the two regions being separated by a vertical line, as shown here. Below, we show the state of affairs during execution of the call on method `getC` in the program above.




3.1 The **new** Expression

The expression `new C(...)` presents problems at first, because the process of cre-

ating and initializing an object is rather strange. The following description of evaluation of this expression helps clear up the confusion. To evaluate `new C(...)`:

1. The beginning of this expression, `new C`, says to create a new folder (or instance) of class `C`, including writing a new name on the tab of the folder.
2. The end of this expression, `C(...)`, is a call on the constructor `C` that appears in the newly created folder, so execute this call; the frame's scope box will contain the name of the folder created in step 1.
3. Yield as the value of the expression the name of the newly created folder.

Evaluation by hand of a few **new** expressions will make clear how the argument values in a new expression get passed first to the parameters and then (in most constructors) to the fields within the newly created folder.

### 3.2. Finding a Target

We discuss determining targets —the variables or methods that are referenced within a method body, e.g. identifiers like `b` and calls like `f(...)`, as well as more complex references like `C.b`, `d.b`, and `d.m(...)`. When reading this section, please remember that the rules for determining targets are not given to the students all at once; pieces of it emerge as new concepts are taught.

The essential ideas can expressed as follows:

1. For a variable `C.v` or method call `C.m(...)`. Look in `C`'s file drawer.
2. For a variable `v` or method call `m(...)`. Look in the active frame. If it is not there, use the inside-out rule to find it —the scope box of the frame gives the surrounding class or object to look in; when looking in an object for it, the overriding rule indicates to start at the bottom and look upward.
3. For a variable `ob.v` or method call `ob.m(...)`. Find the object named by `ob`, using rule 2; then look for `v` or `ob` in the object; the overriding rule indicates to start at the bottom and loop upward.

Unfortunately, these simple rules do not work in some infrequent cases. Shadowing, the redefinition in a subclass of a private method of the superclass, a reference `ob.s` where `s` is a static variable —such exceptional cases makes the rules for finding a target more complicated. We suggest using the above, simple, rules in the first course, saving the more complicated but correct ones for the second course.

Below, we present the correct rules. The only assumption we make is that the only two access modifiers used are **public** and **private**. We have found that the other modifiers merely confuse students and prevent simple guidelines like "make all instance variables **private**". Further, as long as only the default package is used, there is no difference between **public** and the other access modifiers in the code that students write. The target-finding rules can be modified to incorporate other modifiers at a later time.

Our rules use the notion of the *enclosing scope*, which is the next bigger scope; it is defined as follows:

1. For a frame for a call on a static method. The enclosing scope of the frame is the class named in the scope box of the frame.
2. For a frame for a call on an instance method. Suppose the scope box of the frame contains the name `ob`. The enclosing scope is a pair consisting of object `ob` together with the class named by the type of `ob`; this is the apparent view of the object.
3. For a class. The enclosing scope is its superclass (class `Object` if none).

4. For a partition in a folder (object). The enclosing scope is a pair consisting of the inherited partition of the object (the partition above it), if there is one, and the class given by the subclass type (given in the upper right subbox of the partition).

We now give the rules for finding the target for a reference.

A reference `C.b`, where `C` is a class name. To find the target,
1. Look in the file drawer (or class box) for `C`; if it is not there,
2. Look upward through `C`'s enclosing scopes.

A variable name `v`. To find the target,
1. Look in the active frame; if not found,
2. Look upward through the frame's enclosing scopes.

A variable reference `e.b`, where `e` is not a class name. To find the target,
0. Evaluate `e`; its value will be the name of an object, and it will have a type `T`.
1. Look in that object, in its `T` partition (the apparent view) and then in the enclosing scopes of its `T` partition.

A method call `m(...)`, where `m` is an identifier. To find the target,
1. Look in the active frame; if not found,
2. Look upward through the frame's enclosing scopes; then
3. If the method found is a public instance method, look upward from the bottom of the object in which it appears —this is the overriding rule of Sect. 2.2.

A method call `e.m(...)`, where `e` is not a class name. To find the target,
0. Evaluate `e`; its value will be the name of an object, with some type `T`.
1. Look in that object, in its `T` partition (the apparent view) and, if necessary, in the enclosing scopes of the `T` partition; then,
3. If the method found is a public instance method, look upward from the bottom of the object in which it appears —this is the overriding rule of Sect. 2.2.

## 4. WHAT TO TEACH WHEN

In our introductory courses, we teach OO concepts early, for two reasons. First, almost every line of Java has something to do with a class or object. The sooner OO concepts are introduced, the sooner the mysteries of Java notation can be unveiled. More importantly, OO concepts have to be put to use before they are fully understood, and teaching them near the end of the course does not provide opportunity to put them to use. When OOP is taught last, students come out with a purely sequential programming mentality; they don't think easily in OOP terms. Finally, teaching OOP early allows one to make more use of the Java API classes throughout the course.

We have found it effective to teach concepts in the following order, which is advocated in our introduction-to-programming livetext, *ProgramLive* [1].

1. Method calls. We discuss method calls (but not frames), using a small program that draws a circle, rectangle, and text in a graphics window. Students become familiar with executing a sequence of statements. They change arguments of calls and add a few other calls to draw other circles, rectangles, etc. (and see the results of execution on

their monitor). They learn to read specs of methods and write method calls based on them. They learn that to determine what a call does, make a copy of the method spec and replace occurrences of parameters in it by the corrresponding arguments of the call. They become familiar with simple expressions and running Java programs.

2. Simple method bodies. We introduce `System.out.println` , if-statements, and return statements in writing method bodies (we deal only with static methods). We introduce the frame for a method call, which at this point has only its name and parameters, and have students execute method bodies as follows:

(a) Evaluate the arguments of the call, from left to right, pushing their values onto the call stack (as boxes, named with the corresponding parameter name).

(b) Draw the rest of the frame on the call stack, filling in the name of the method.

(c) Execute the method body. Look in the active frame for parameter values.

(d) Pop the frame from the call stack. For a function, push the value to be returned onto the call stack.

(e) Continue execution at the place given by the line counter in the active frame.

6. Local variables. We introduce local variables to hold intermediate results during execution of the method body. We extend execution of a call to include local variables.

7. The class. We introduce the class as a file drawer that holds static methods. We present class `java.lang.Math` and its static methods. We explain how to access the API specifications for class `Math`. Static variables, especially constants (with modifier **final**), are a byproduct of this discussion.

8. Folders (objects). We discuss classes thoroughly. We introduce the scope box of the frame and show how it is used. Constructors and constructor calls are handled.

9. Subclasses. This explanation of subclasses includes overriding and casting. To simplify the presentation, we do not ban shadowing of variables.

10. Loops. Loops are difficult for students to grasp. Rarely do they write loops simply and correctly. Teaching OO first gives the students time to mature somewhat before they get to loops.

11. Arrays. You can do a lot of OOP without arrays. Class Vector can be introduced early, if desired. Strings can also be used for various purposes.

## 5. INNER CLASSES

In Java, a static class defined inside another class is called a *nested class*; a nonstatic class defined inside another class is called an *inner class*. Inner and nested classes don't belong in the first programming course, but they should be taught in the second course because of their usefulness in structuring programs.

An example of an inner class is `HashIterator` within class `HashMap` in package `java.util`. It is *inner* because it refers to nonstatic fields of `HashMap`. It is private, and within `HashIterator`, for two reasons: (1) There is no need for other classes to reference it; this is a good use of information hiding and a good software engineering technique. (2) `HashIterator` references private nonstatic fields of `HashMap`.

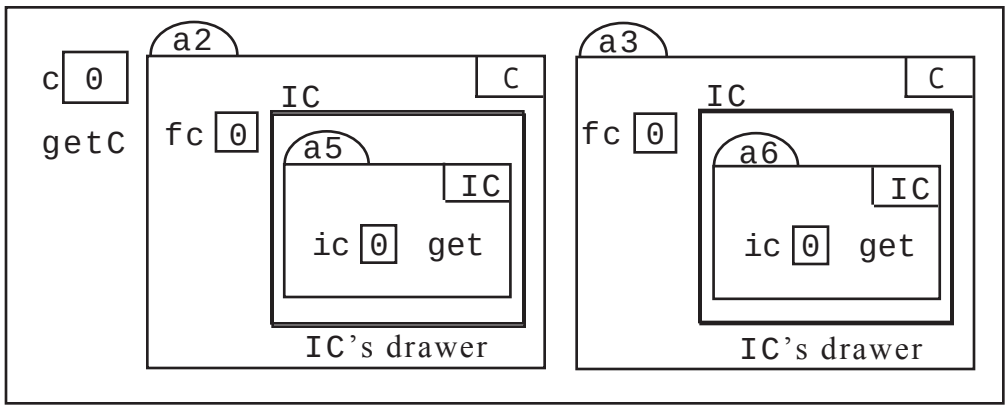
According to the rules we have given so far, a file drawer for an inner class appears inside each instance of the outer class —one file drawer gets crammed inside another. Consider this class:

```
public class C {
   public static int c;
```

```
    public static int getC()
       { return c; }
    private int fc;

    public class IC {
       private int ic;
       public int get()
          { return ic+fc; }
    }
}
```

Below, we show C's file drawer, with two instances (folders) of class C in it. Each one contains a file drawer for class IC , since IC is defined as a nonstatic component of C.




C's file drawer

According to the inside-out rule (page 3), methods in folder a5 can reference:

- Field ic and method get of a5.
- Field fc and class IC of a2.
- static field c and static method getC in C's file drawer.

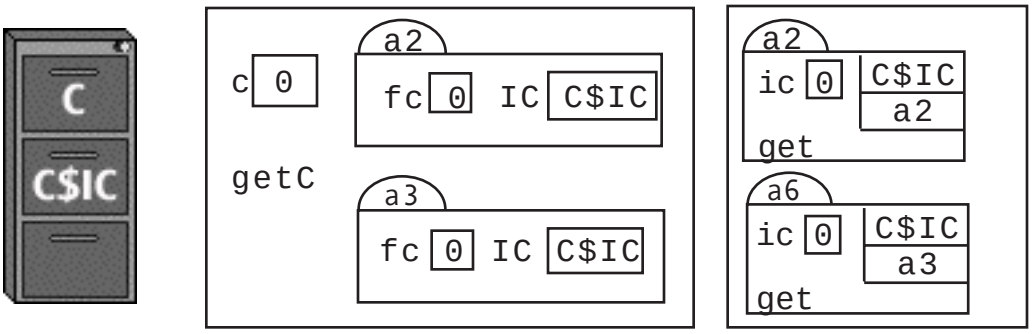
For our algorithm for finding the target of a reference to work correctly, we have to define the *enclosing scope* of an inner class: this is the enclosing scope of the class together with the object in which inner class appears.

The diagram of C's file drawer (above) is useful in providing understanding of inner classes. It helps indicate exactly what an instance of an inner-class (e.g. a5) can reference. It also helps explain why inner classes are useful.

In this case, a file drawer for IC must appear within each instance of C so that its instances (e.g. a5 ) can reference the fields of those instances of C.

Java itself uses a flattened version of the diagram above, corresponding to the flattened view shown below. There is only one file drawer for class IC, because it does not (can not) have static elements. All the folders in all the different instances of IC can go in the single drawer (they have distinct names). However, each instance of a class IC needs a scope box, which contains the name of the instance of C in which the instance belongs. We draw this scope box as a second box in the upper right of the folder. For example, in the diagram above, the inner view, folder a5 appears in the file drawer in instance a2. Therefore, in the diagram below, the scope box of folder a5 contains the name a2. This scope box can be used to construct the conventional view from the flattened view. Note that C$IC's file drawer also has a scope box, which contains the name C of the class in which IC is defined.

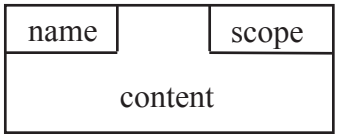
Java actually creates a class named C$1 (instead of C$IC), which you can see in the directory on your hard drive where the .class files are stored (or in the jar file).




C's file drawer C$IC's file drawer

## 6. A MODEL OF EXECUTION THAT IS MORE IN TERMS OF MEMORY

We can describe the execution model without using file drawers, talking about memory locations, addresses, and pointers. First, describe memory as a list of locations, each of which has a number, called its address —much like houses on a street. When a variable is declared, the computer associates it with a particular location, possible as part of an object or class. The value of the variable is stored in that assigned location. When an object is created, its components are stored in contiguous memory locations. The value of a reference variable is the address of the memory location where the object is stored. Retrieving the value of a reference `v.var` requires first retrieving the address in `v` and then retrieving the value of `var` from the object at that address.

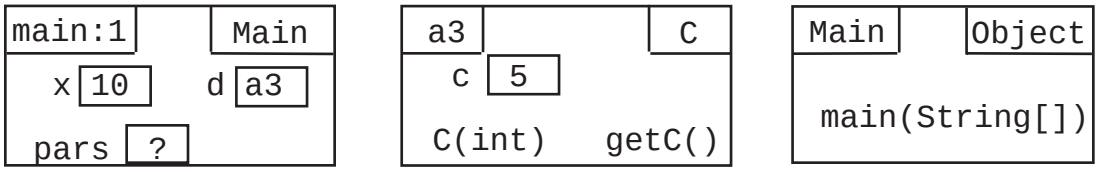



We introduce a standard way of drawing information. Every object, class, and frame is drawn as shown to the left. Method frames contain local variables and parameters; objects, instance variables and methods; and classes, static variables and static methods. Consider the following code.

```
public class Main {
   public static void main
          (String[] pars) {
      int x= 10;
      C d= new C(x);
      x= d.getC();
   }
}
```

```
public class C {
   private int c;
   public C(int pc)
      { c= pc; }
   public int getC()
      { return c; }
}
```

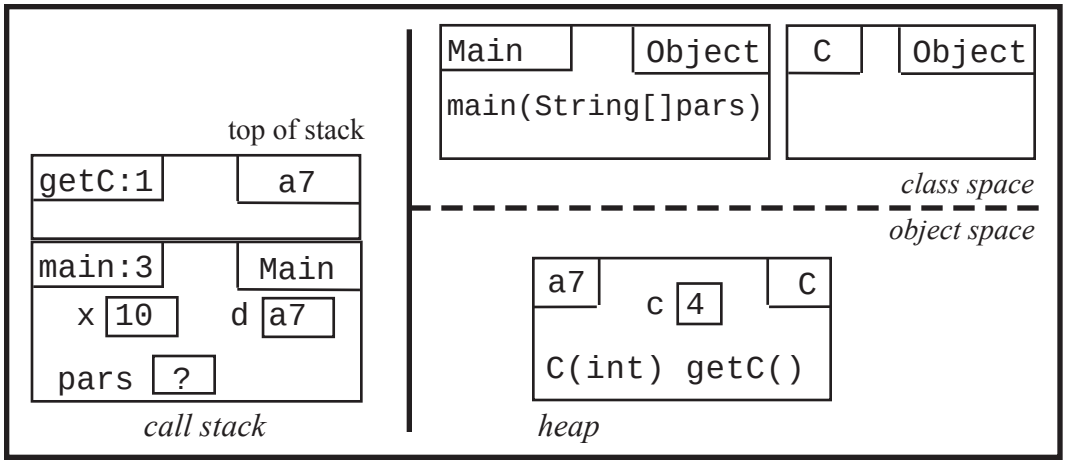
Below are examples of a method frame, an object, and a class. These pictures would be drawn during a trace of the previous program.




a frame an object a class

To ensure that students do not confuse frames, objects, and classes, we draw memory with three regions: frames are drawn in the method space, objects in the object space, and classes in the class space (see below). This flat view is easy to draw. It clear-

ly separates the stack from the heap (the class and object spaces combined) and prevents students from writing static information inside objects or thinking that local variables are persistent.




The rules for executing method calls and finding targets, given in Sect. 3, hold with this model as well.

ACKNOWLEDGEMENTS

We thank Allan D. Jepson for his help and the referees for their advice.

REFERENCES

[1] Gries, D., Gries, P. *ProgamLive*. John Wiley, NY, 2000. This text teaches programming using Java. It comes on a CD and has over 250 2-3-minute recorded lectures with synched animation. For more info visit http://www.wiley.com/college/gries.
[2] Gries, P., Hall, P., Gries, D. *Companion to ProgramLive*. John Wiley, NY, 2001.