TRACING CODE:
THE MEMORY MODEL

Areas of Memory

There are two areas of computer memory for a running program:

Run-Time Stack: (a.k.a. *call stack*):

- Holds information that is local to method calls, like parameters, local variables, and which line of code is being executed.
- When a method terminates, all this information is erased.

Heap:

- Holds longer-lived information, like:
  - objects and their contents (anything created with `new`)
  - static information.

The memory model traces how the computer uses these two areas while running a program.*

* *The memory model rules deal only with running code.* They do not describe what happens at compile time, such as figuring out whether a private variable can be accessed.

Stack:
Method Space
(the run-time stack)

One box ("stack frame" or "method frame") for each running method.

Each frame contains that method's parameters and local variables.

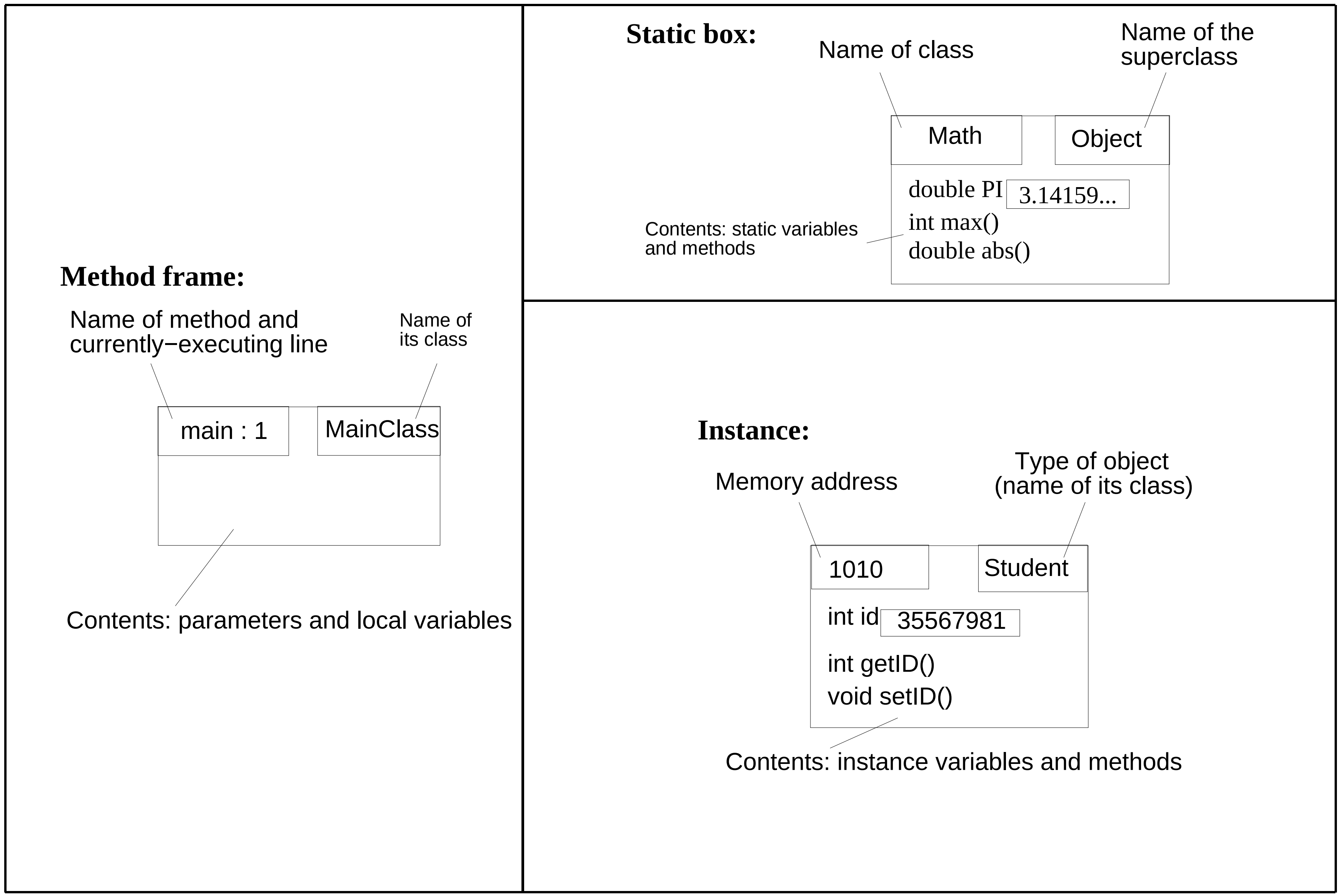
Heap: Static Space

One box for each class, containing static variables and static methods. (These methods are available, but not running.)

No new boxes are created here during program execution.

Heap: Object Space

One box for each existing object, containing instance variables and instance methods. (These methods are available, but not running.)


Method frame:
Name of method and
currently−executing line
Name of
its class
main : 1
MainClass
Contents: parameters and local variables
Static box:
Name of class
Name of the
superclass
Math
Object
double PI
3.14159...
int max()
double abs()
Contents: static variables
and methods
Instance:
Memory address
Type of object
(name of its class)
1010
Student
int id
35567981
int getID()
void setID()
Contents: instance variables and methods

Tracing Program Execution

1. **Load the classes and interfaces.**

   Load each class and interface by drawing its static space.

   Follow the rules on the upcoming slides.

2. **Call method main.**

   Begin execution by tracing a call to method main().

3. **Trace each statement line by line.**

   Follow the rules on the upcoming slides.

Step 1: Loading the Classes and Interfaces

For each class or interface:

1. Draw a box in the static space.
2. Write the name of the class or interface in the top-left corner.
3. Write the name of the parent class in the top-right corner, along with any interfaces that the class implements.
4. Draw:
   - static variables: type, name and value
   - static methods: return type and name

Note that only one copy of each static member exists, no matter how many objects are created.

Example: Trace the loading process for this program . . .

TestFrac program*

```
public class TestFrac {
    public static void main(String[] args) {
        Frac f1 = new Frac(3, 4);
        Frac f2 = new Frac(2, 3);
        Frac f3 = new Frac(1, 2);
        Frac f4 = Frac.max(f1, Frac.max(f2, f3));
    }
}

public class Frac {
    private int numer, denom;
    private static int numCreated;

    public Frac(int n, int d)
      {  numer = n;  denom = d;  numCreated++;  }

    public static Frac max(Frac a, Frac b) {
        int aSize = a.numer*b.denom;
        int bSize = b.numer*a.denom;
        if (aSize > bSize)  return a;
        else  return b;
    }

    public Frac mult(Frac f) {
        return new Frac(this.numer * f.numer, this.denom * f.denom);
    }

    public String toString()
      {  return numer + "/" + denom;  }
}
```

*Apologies for the names: they are abbreviated to make the code fit on one page.

Step 3: Tracing statement execution

These are the types of statements we have to trace.

Statement type	Syntax
method call	`expression.methodname(args);` (args is a comma-separated list of expressions) **Example:** `s.substring(3,5);`
declaration	`type identifier;` **Example:** `String s;`
assignment	`identifier = expression;` **Example:** `t = -55;`
initialization	`type identifier = expression;` (initializations combine declarations and assignment statements) **Example:** `int i = 3;`
return	`return expression;` **Example:** `return f();`

Example

```
class Simple {
    public static int zonkest(int one, int two) {
        if ((one > 0) && (one < two))
            return one;
        else
            return two;
    }

    public static void main(String[] args){
        int i = 7;
        int j = 4;
        int k = -2;
        int l = zonkest( (i+j)/k, j*k );
    }
}
```

A very complex method call

```
zonkest( Math.max(s.length(), t.length()+1),

         ((String)(v.elements().nextElement()))
                        .length()
       );
```

Tracing Rules

Method call:

1. In the code for the method call, label the expressions with Roman numerals to indicate the order in which they will be evaluated.

2. In order, evaluate each argument and draw a box on the top of the stack to hold the argument value.

3. Draw a frame for the method on top of the stack; include the argument boxes from step 2 inside the new frame.

4. Write the method name in the top-left corner and the method scope in the top-right corner.*

5. Any argument values will be on top of the method stack from step 1. Rename the box for each value to the corresponding parameter name.

6. Write :1 (the line number) after the method name.

7. Execute the method line-by-line, incrementing the line number.

*The method scope is the address of an object if the method is non-static, and is the name of a class if the method is static.

Declaration:

In the current frame, write the variable type and name, and draw a box to hold the value.

Assignment:

1. Evaluate the expression on the right side of =.
2. Write the result in the variable referred to on the left side.

Do *not* create a new box.

Initialization: Do the declaration and then the assignment (as above).

return: Evaluate the expression and replace the current method frame with the result value.

Tracing statements involves evaluating expressions (inside-out and left to right).

“new” expression (special because it creates an object):

1. Draw a new object in the object space.

   Use a stack of boxes to represent the object’s class and its ancestors in the inheritance hierarchy.

   For each box:

   - Write the class name in the top-right corner, along with any implemented interfaces.
   - Draw:
     - instance variables: type, name and default value
     - instance methods: return type and name

2. In the topmost box, write the address of the object in the top-left corner.
   Represent the address with an arbitrary four-bit number (e.g., `0010`, `1010`).

3. Execute the constructor call.
   The constructor’s scope is the new object.

4. When the constructor is done, the value of the `new` expression is the address of the new object.

Example: `Frac f1 = new Frac(3, 4);`

Special cases with “new”

You can create a String object without saying “new”.
Example:

```
String s = "Wombat";                // Shorthand.
String s = new String("Wombat");  // What it means.
```

What about drawing an instance of a class that you didn’t write, such as String?

- You probably don’t know what the instance variables are.
- Yet you need to keep track of the contents of the object somehow.

Just make up a sensible notation.

Examples:

```
String s = new String("Wombat");
Integer i = new Integer(27);
Vector v = new Vector();
v.addElement(s);
v.addElement(i);
```

Simplifications

When tracing, simplifications such as these may be justified:

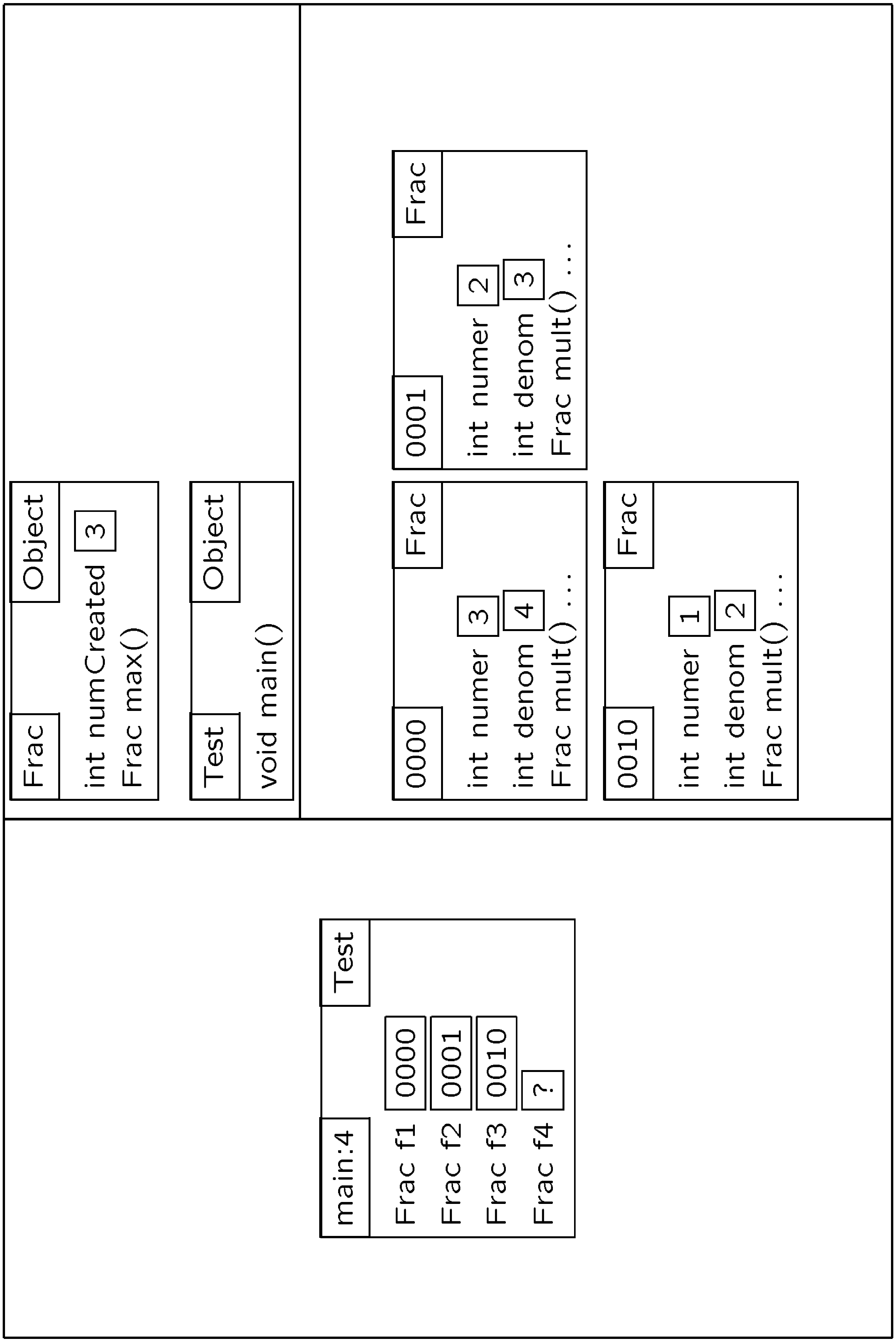
- If a class contains nothing static, omit its static box.
- When drawing an object, include boxes for only those ancestor classes that you wrote yourself.
- Omit variable types.

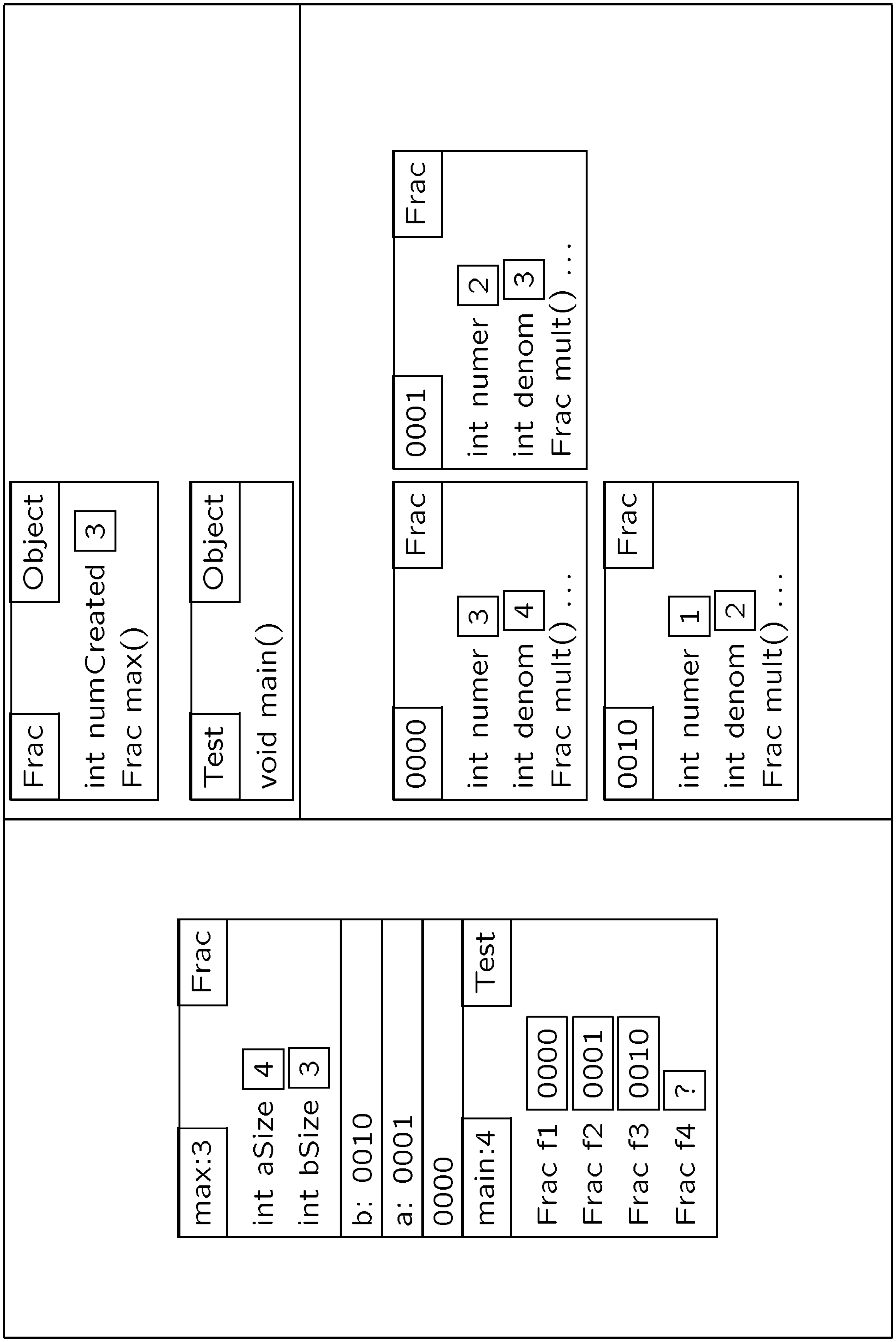
Make simplifications only where you are confident about the code. In the places where you are unsure, include all the detail.

TestFrac Program

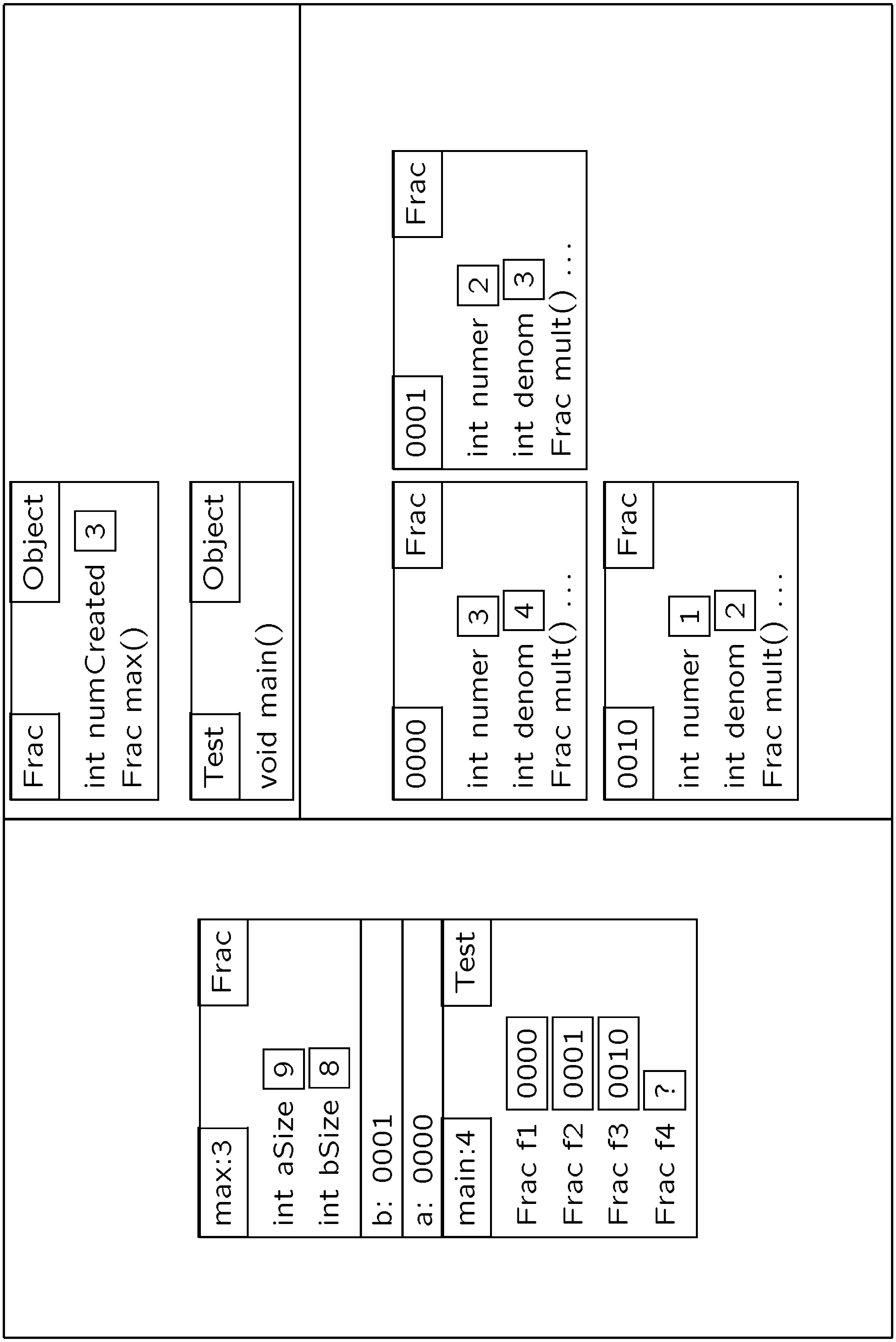
Now trace this fully.

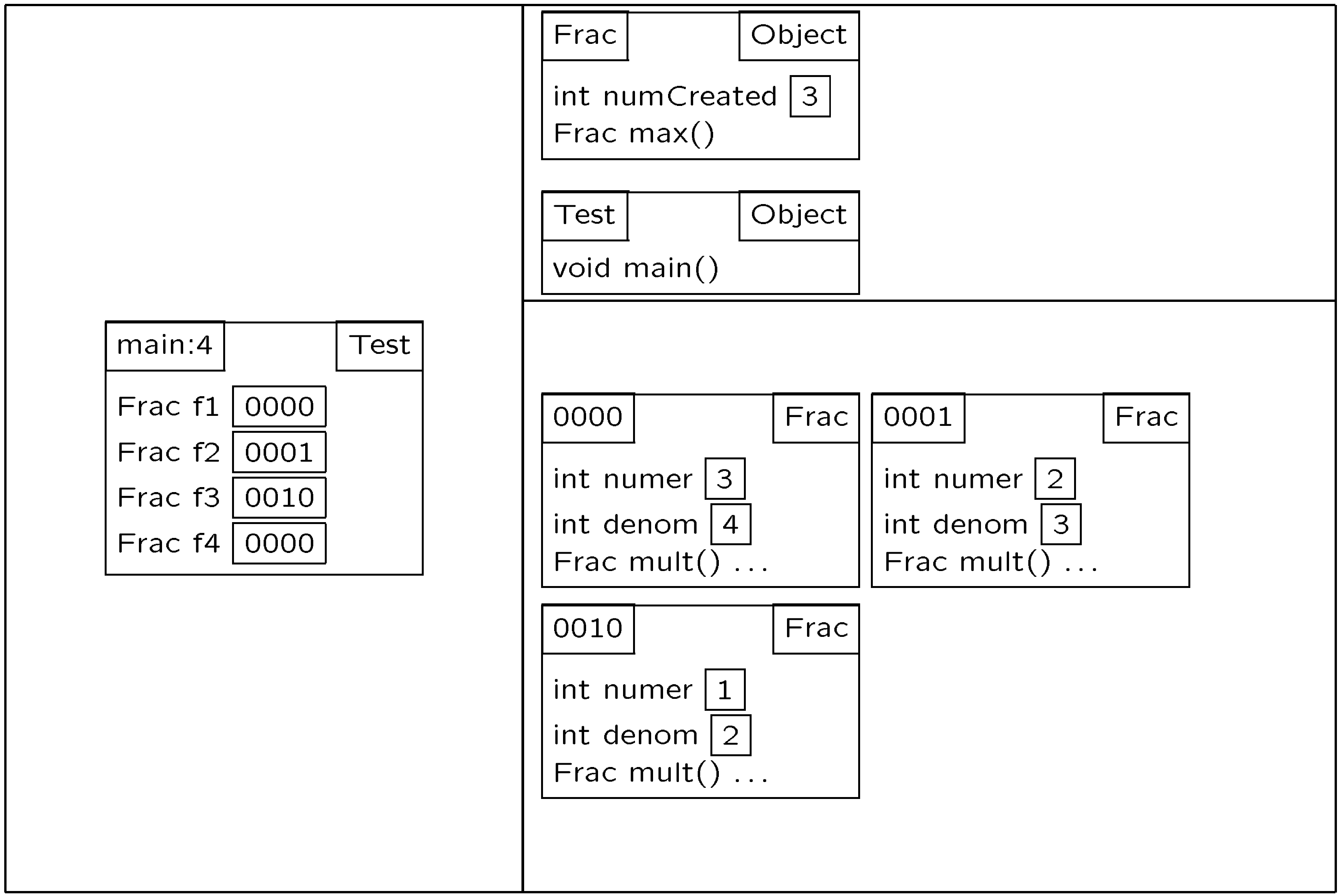
```
public class TestFrac {
    public static void main(String[] args) {
        Frac f1 = new Frac(3, 4);
        Frac f2 = new Frac(2, 3);
        Frac f3 = new Frac(1, 2);
        Frac f4 = Frac.max(f1, Frac.max(f2, f3));
    }
}

public class Frac {
    private int numer, denom;
    private static int numCreated;

    public Frac(int n, int d)
      {  numer = n;  denom = d;  numCreated++;  }

    public static Frac max(Frac a, Frac b) {
        int aSize = a.numer*b.denom;
        int bSize = b.numer*a.denom;
        if (aSize > bSize)  return a;
        else  return b;
    }

    public Frac mult(Frac f) {
        return new Frac(this.numer * f.numer, this.denom * f.denom);
    }

    public String toString()
      {  return numer + "/" + denom;  }
}
```


Frac
Object
int numCreated 3
Frac max()
Test
Object
void main()
0000
Frac
int numer 3
int denom 4
Frac mult() ...
0001
Frac
int numer 2
int denom 3
Frac mult() ...
0010
Frac
int numer 1
int denom 2
Frac mult() ...
main:4
Test
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 ?



Frac
Object
int numCreated
3
Frac max()
Test
Object
void main()
0001
Frac
int numer
2
int denom
3
Frac mult() ...
0000
Frac
int numer
3
int denom
4
Frac mult() ...
0010
Frac
int numer
1
int denom
2
Frac mult() ...
max:3
Frac
int aSize
4
int bSize
3
b: 0010
a: 0001
0000
main:4
Test
Frac f1
0000
Frac f2
0001
Frac f3
0010
Frac f4
?


Frac
Object
int numCreated 3
Frac max()
Test
Object
void main()
0000
Frac
int numer 3
int denom 4
Frac mult() ...
0001
Frac
int numer 2
int denom 3
Frac mult() ...
0010
Frac
int numer 1
int denom 2
Frac mult() ...
max:3
Frac
int aSize 9
int bSize 8
b: 0001
a: 0000
main:4
Test
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 ?


main:4
Test
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 0000
Frac
Object
int numCreated 3
Frac max()
Test
Object
void main()
0000
Frac
int numer 3
int denom 4
Frac mult() . . .
0001
Frac
int numer 2
int denom 3
Frac mult() . . .
0010
Frac
int numer 1
int denom 2
Frac mult() . . .

What the heck does this print?

```
class Tricky {
    public static void main(String[] args) {
        A a = new A();  B b = new B();
        I i = (I) b;  P p = (P) i;
        A.sm();  a.m();
        b.sm();  i.m();   p.m();
        b.m().sm();
    }
}
```

```
public interface I {
  static final int ANSWER = 42;
  public P m();
}

// Parent class P
public class P implements I {
  static int sv = 9;
  int v = 8;

  public static void sm() {
    System.out.println(
      "P: sm(): sv = " + sv);
  }

  public P m() {
    System.out.println("P: "
      + sv + " X " + v + " = "
      + ANSWER);
    return this;
  }
}
```

```
// Sibling classes A and B

public class A extends P { }


public class B extends P {
  static int sv = 6;
  int v = 7;

  public static void sm() {
    System.out.println(
      "B: sm(): sv = " + sv);
  }

  public P m() {
    System.out.println("B: "
      + sv + " X " + v + " = "
      + ANSWER);
    return this;
  }
}
```

Good use of static information

Even now, the `Tricky` program is hard to follow. It is easier to trace code that obeys the following stylistic rule.

Rule: When accessing a static member of a class, always use that class name.
Examples:

```
// Bad:          // Better:
b.sm();          P.sm();
A.sm();          P.sm();
```

Exception: When the static information is in the current class, you may omit the class name.
Example:

```
// Inside method m() of class P, we can access
// P's method sm() this way:
P.sm();

// It's also okay to access it like this:
sm();
```

The remaining slides assume that code follows this rule.

Tracing Expressions

Some expressions in a program

- refer to a variable
  - Examples: `I.ANSWER`, `P.sv`, and `b.v`
  - We need to know exactly which variable such an expression refers to before we can find its value.
- or refer to a method
  - Examples: `P.sm()`, and `b.m()`.
  - We need to know exactly which method such an expression refers to before we can call it.

The variable or method referred to by an expression is called its **target**.

Finding the Target

One of the problems with tracing a program like `Tricky` is that the target of some expressions is not obvious! We need a technique.

Expressions of the form *e.mem*

If we see an expression of the form *e.mem*, such as,

```
(s.foo(3)).count
```

we know that:

- *e* is itself an expression.

  If it is the name of a class, *mem* is static.

  Otherwise, it evaluates to the address of a box on the heap. In that case, it has a type, which identifies a part of that box.

- *mem* is a variable or method call.
  That variable or method is our target.

Algorithm

To find the target of a compound reference *e.mem*:

- if *e* is a class or interface name, *mem* is static:
  Look for *mem* in the static box for *e*.
- if *e*'s value is the address of an object and *mem* is a variable:
  Find *e*'s type T and look for *mem* first in the T part of that object. If *mem* is not there, go up the inherited boxes until you find it.
- if *e*'s value is the address of an object and *mem* is a method:
  Look for *mem* first in the **bottom** part of that object, regardless of *e*'s type. If *mem* is not there, go up the inherited boxes until you find it.

Special Cases

To find the target of a simple variable reference *v*:

- Look for *v* in the topmost stack frame. If *v* is there, that's the target (and *v* is a local variable). If *v* is not there, treat the expression as if it were `this.`*v*.

To find the target of a simple method reference *p*(*args*):

- Treat the expression as if it were `this.`*p*(*args*).

We've seen that the target of *e*.*m* depends on *e*'s type. Casting can affect that ...

Casting

The type of an object

The type of an object is the most specific classname in it.
Example: When we say "`new A();`" we construct an object that has an `A` part and a `P` part, but the object's type is `A`.

Widening is automatic

The rules we've just seen for finding a target allow us to automatically go up to the higher and more general sub-parts of an object.
Examples:

```
// The object has a B part and a P part, and its type
// is B.  This is widened to match fum's type, P:
P fum = new B();
// v is found up in the P part of this object:
A fee = new A();
fee.v = 21;
```

Same as automatic widening with primitives.
Example: `double d = 3;`

Narrowing requires a cast

To go down to the lower and more specific sub-parts of an object, we must explicitly cast.

Example: Suppose class `B` also had a variable `t` that class `P` lacked.

```
// The object has a B part and a P part, and its type
// is B:
P fum = new B();

// "((B)fum)" has type B, so we look in the B part of
// the object and work up.  t() is found in the B part:
((B)fum).t = 7;
```

This is the same as explicit narrowing with primitive variables.
Example: `int = (int) 4.27;`

Precedence

The precedence of the dot operator "." is higher than the precedence of the casting brackets. So this won't work:

```
(B)fum.t = 7;
```

That's why we need extra brackets:

```
((B)fum).t = 7;
```

What we can cast to

We can cast to any type that appears in the object: the class of the object, any superclass or subclass, and any interface that any class of the object implements.

Although we can, we never need to cast to a superclass, because of widening.

What casting does

Casting changes the type of an expression. It does not change the address of an object or the type of an object.
Example:

```
B b = new B();       // The new object never moves and
                     // always has type B.
P p = b;             // The expression "p" has type P.
B otherB = (B) p;    // But the expression "(B) p"
                     // has type B.
```

More examples with casting

Exercise: For each assignment below, explain why a cast is or is not required.

```
Object o = b;    // Cast not required.
p = (P) o;       // Cast required.
I i = p;         // Cast not required.
b = (B) i;       // Cast required.
```

Shadowing and Overriding

In object oriented languages it is possible to *override* methods:

- If there are several instance methods with matching names and arguments in an object then, no matter what type of reference is used, the bottom-most method body in the object is invoked.
- For example, `b.m()`, `((P) b).m()`, and `((I) b).m()` all refer to the same method body, namely the `m()` in the class B part of the object.

Instance variables behave differently:

- An instance variable is said to *shadow* a variable of the same name in a superclass. Unlike method overriding, the shadowed variable in the superclass can be referenced by casting, as in `((P) b).v`.

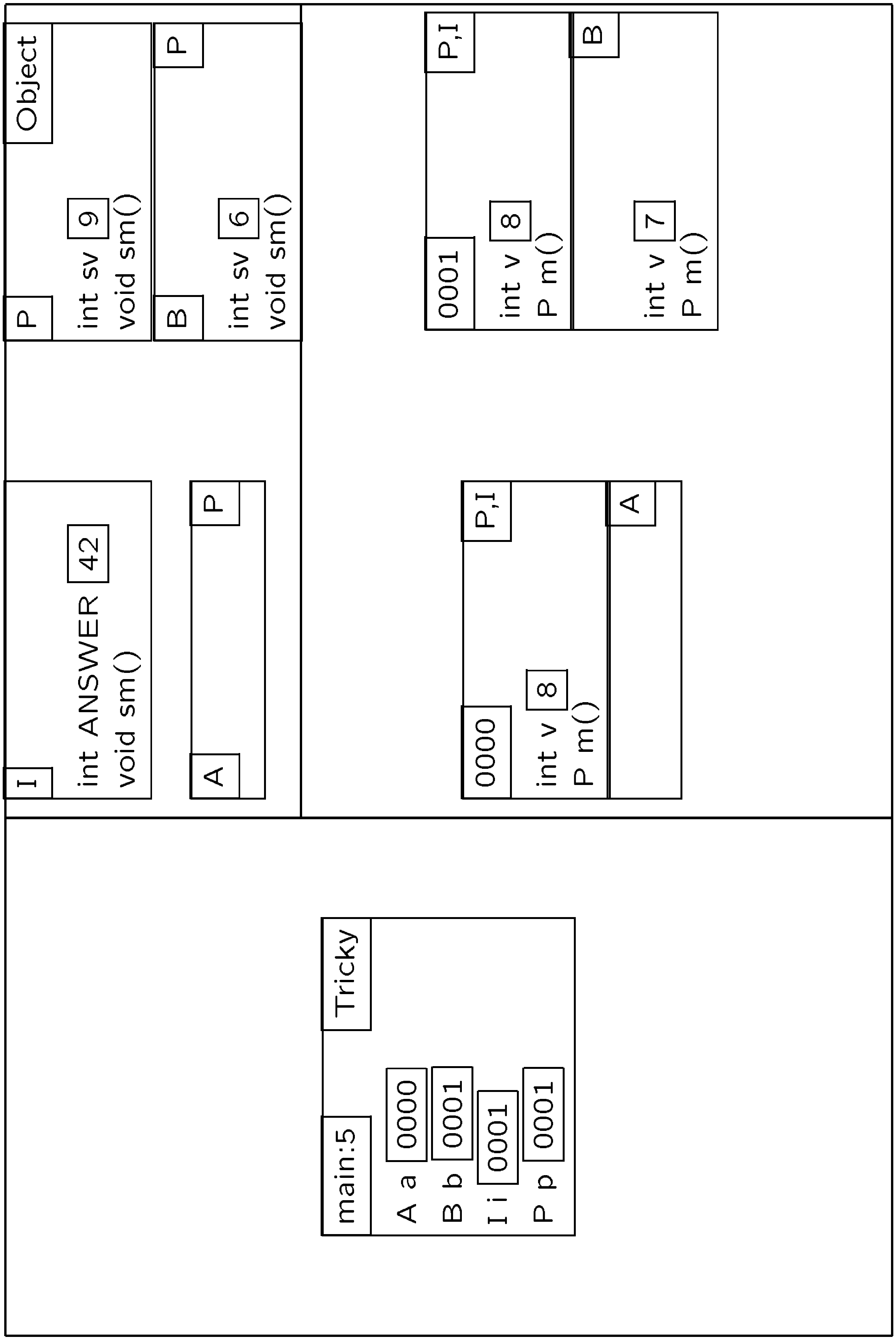
Targets in our Tricky Program

Remember that to find the target of an expression r.v or r.m(), we need to know not only the value of r, but its type.

Expression	Type of r	Value of r	Target var	Value of the expn
`b.v`	B	0101	`v` in `B` at 0101	7
`a.v`	A	0100	`v` in `P` at 0100	8
`((P) b).v`	P	0101	`v` in `P` at 0101	8
`b.sv`	B	n/a	`sv` in `B`	6
`a.sv`	A	n/a	`sv` in `P`	9
`((P) b).sv`	P	n/a	`sv` in `P`	9

Expression	Type of r	Value of r	Target method
`b.m()`	B	0101	`m()` in `B` at 0101
`a.m()`	A	0100	`m()` in `P` at 0100
`((P) b).m()`	P	0101	`m()` in `B` at 0101
`((I) b).m()`	I	0101	`m()` in `B` at 0101
`b.sm()`	B	n/a	`sm()` in `B`
`a.sm()`	A	n/a	`sm()` in `P`
`((P) b).sm()`	P	n/a	`sm()` in `P`

Question: Which expressions above are disallowed by our style rule for static variables?


Object
P
int sv 9
void sm()
P
B
int sv 6
void sm()
I
int ANSWER 42
void sm()
P
A
P,I
0001
int v 8
P m()
B
int v 7
P m()
P,I
0000
int v 8
P m()
A
Tricky
main:5
A a 0000
B b 0001
I i 0001
P p 0001

Keywords `this` and `super`

Now it's easy to understand `this` and `super`.

`this`:

- Always refers to the address in the top-right of the top stack frame.
  (If it contains a class name instead of an address, using `this` is illegal.)
- Its type is the part of the object where the method is.

`super`:

- Always refers to the address in the top-right of the top stack frame.
- But its type is one up.
- We can use `super` to get at an overridden method.

Trace the following examples ...

Super example

```
// Suppose class B had this additional method:
public void newMethod() {
    super.m();
}

// Now in Tricky's driver we can say:
B fo = new B();
fo.m();            // Calls the m() in class B.
fo.newMethod();  // Lets us call the m() in P.
```

This example

```
public class TestThis {
    public static void main(String[] args) {
        Top t = new Top(); Bot b = new Bot();
        t.topMeth();
        b.botMeth(); b.topMeth();
    }
}

class Top {
    int v = 3;
    void topMeth() {
        System.out.println("In topMeth: " + this.v);
    }
}

class Bot extends Top {
    int v = 4;
    void botMeth() {
        System.out.println("In botMeth: " + this.v);
    }
}
```