Java Memory Management

Tracing program execution

Trace: To follow the course or trail of.

When you need to find and fix a bug or have to understand a tricky piece of code that your coworker wrote, you have to trace it. There are three ways to trace code:

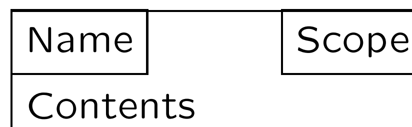
1. Put in a lot of print statements.
2. Use a debugger.
3. Trace by hand.

Many undergraduate students use the first trick to debug and understand code. *Professional programmers almost never do.*

A picture of computer memory

Information about a running program is stored in computer memory. Every object, class, and running method has a separate region of memory to keep track of variable values and other related information.

We represent each region using this picture, called a *memory box*:



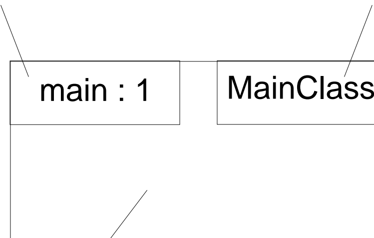
Static information (class boxes) and instance information (object boxes) are stored in the *heap*. Method information is stored in the *run-time stack*.

Stack: method space (contains method boxes)

Method frame:

Name of method and currently–executing line number

Name of class (for static method) or address of object (for instance method)

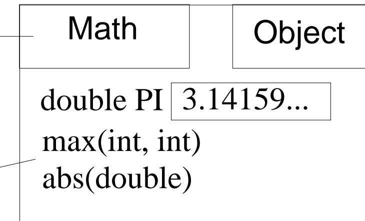


Contents: parameters and local variables

Heap: Static Space (contains static boxes)

Static box:

Name of class



Name of superclass

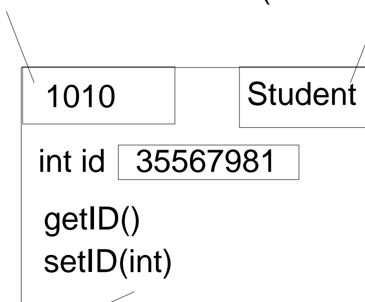
Contents: static variables and static methods

Heap: Object Space (contains instance boxes)

Instance:

Memory address

Type of object (name of its class)



Contents: instance variables and methods

Tracing statement execution

You have an intuitive understanding of how code works: the program is executed line by line, with jumps between methods.

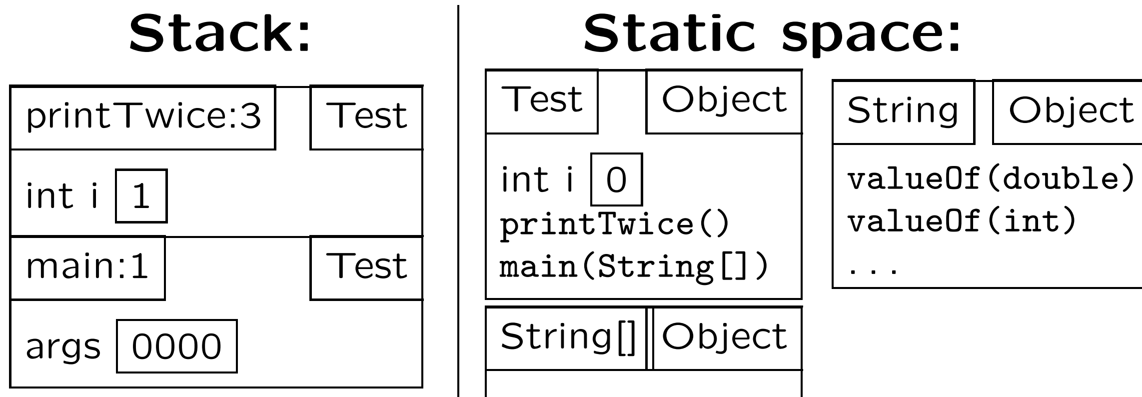
The next several slides describe the rules used by the computer to figure out:

- When memory boxes are created for methods, objects, and classes.
- How assignment statements work.
- How argument values are assigned to parameters.
- How method return values are handled.

Run-time stack example

```
public class Test {
    public static int i = 0;
    public static void printTwice() {
        int i = 1;                         // Line 1
        System.out.println("Hello");   // Line 2
        i = 2;                             // Line 3
        System.out.println("There");   // Line 4
    }
    public static void main(String[] args) {
        printTwice();                      // Line 1
    }
}
```

Here are the method stack and static boxes for this program, after line 2 of `printTwice` has finished, but before line 3 begins. (The object referred to by `args` is not shown.)



Class loading

Classes are *loaded* into memory. We trace this by drawing a box for the class, and write in the static variables and static methods.

In this course:

When tracing in this course, you can just draw the static boxes for them before they start up.

To do this, read through the code and, whenever you see a class name, draw a static box for it.

To find out what really happens, turn the page.

Class loading, in the JVM

The real rules:

Classes are loaded when they are needed. For example, before method `main` is called by the Java Virtual Machine it *loads* (sets up) `main`'s class and all its superclasses.

In general, a class is loaded the first time it is needed, which is either

- when a variable of the class is declared,
- when an object of the class is created, or
- when a static method or variable in the class is used.

When a class is loaded, trace it by drawing a box for the class in the static space.

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

TestFrac program

Exercise: Identify when classes are first loaded. (Don't forget to think about `String`.)

```
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;  }
}
```

Tracing statement execution

These are the types of statements we have to trace.

Statement type	Syntax
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();`
method call	`expression.methodname(args);` (args is a comma-separated list of expressions) **Example:** `s.substring(3,5);`

Tracing Rules

Local var declaration (`type var;`):

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

Assignment (`lhs = rhs;`):

1. Find the target of the variable in the lhs.
2. Evaluate the expression on the rhs.
3. Write the value in the box for the target. (Do *not* create a new box.)

Initialization (`type var = rhs;`):

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

return (`return expr;`):

Evaluate *expr* and replace the top method frame with the value.

Method call:

1. In the code for the method call, label all expressions and subexpressions in the arguments with Roman numerals to indicate the order in which they will be evaluated (inside out, left to right).

2. In order, evaluate the (sub)expressions for each argument and draw the argument values in boxes on the top of the stack.

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.

A simple method call

```
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() );
```

The sequence of method calls is determined by working left to right, inside out.

“new” expression:

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: signature

2. In the topmost box, write the address of the object in the top-left corner.
   Represent the address with an arbitrary hexadecimal number (e.g., `0010`, `19af`).

3. Draw the constructor on top of the runtime stack, with the new object as the constructor’s scope. Execute the constructor.

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

Trace this: `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.

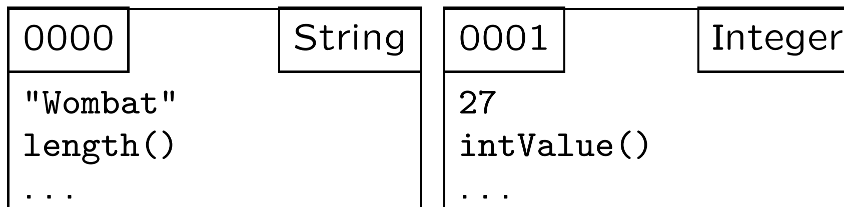
  The next slide contains a few examples.

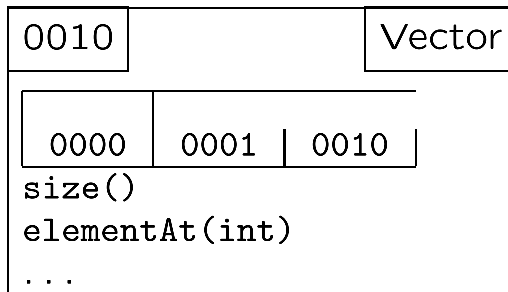
Drawing Java API objects

Trace these examples:

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

Object memory boxes:





Simplifications

When tracing, simplifications such as these are acceptable:

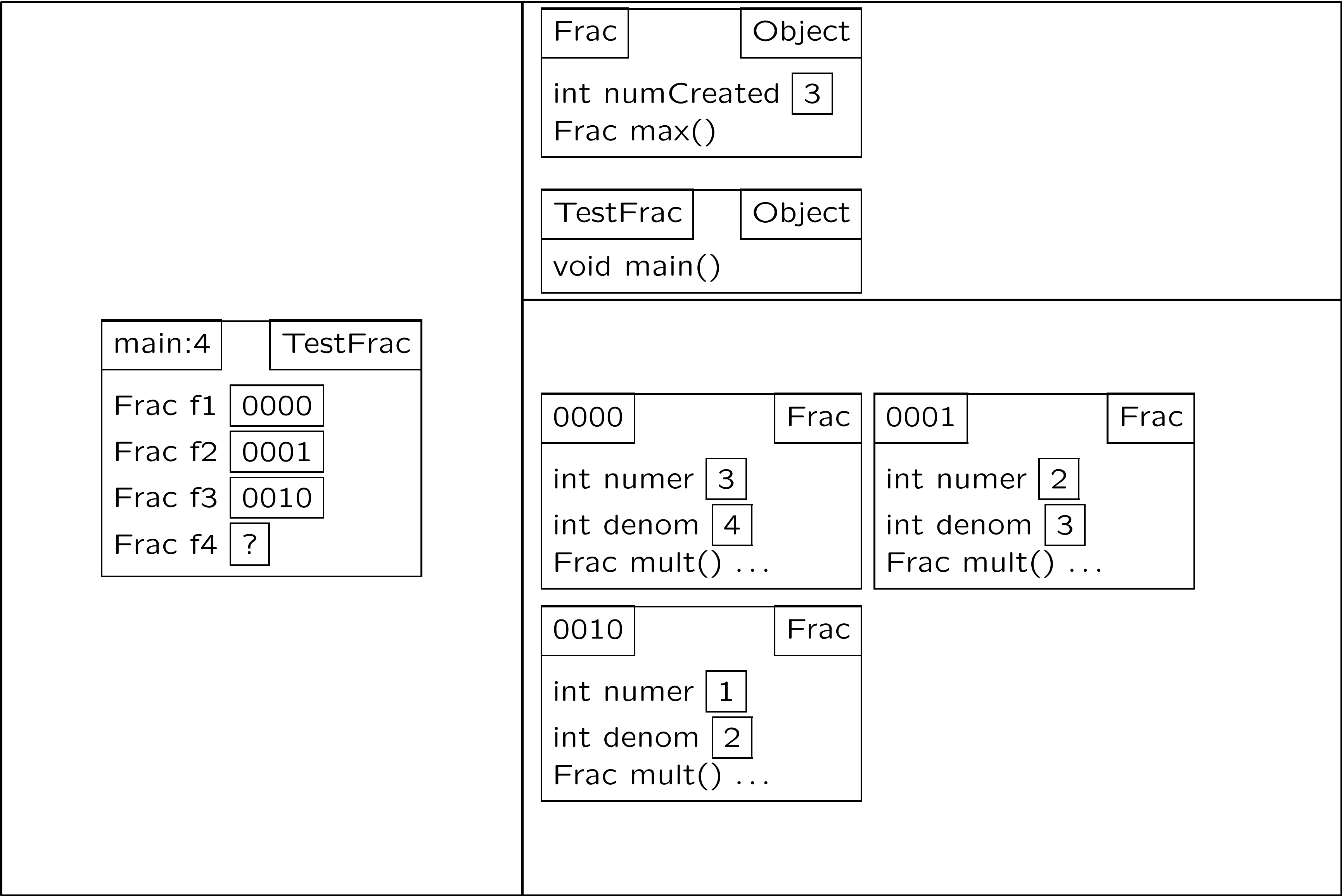
- If a class contains nothing static, omit its static box.
- When drawing an object, include boxes for only those ancestor classes that you wrote yourself. (For example, you can always omit `Object` unless it's used by your code).
- Omit variable types.

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

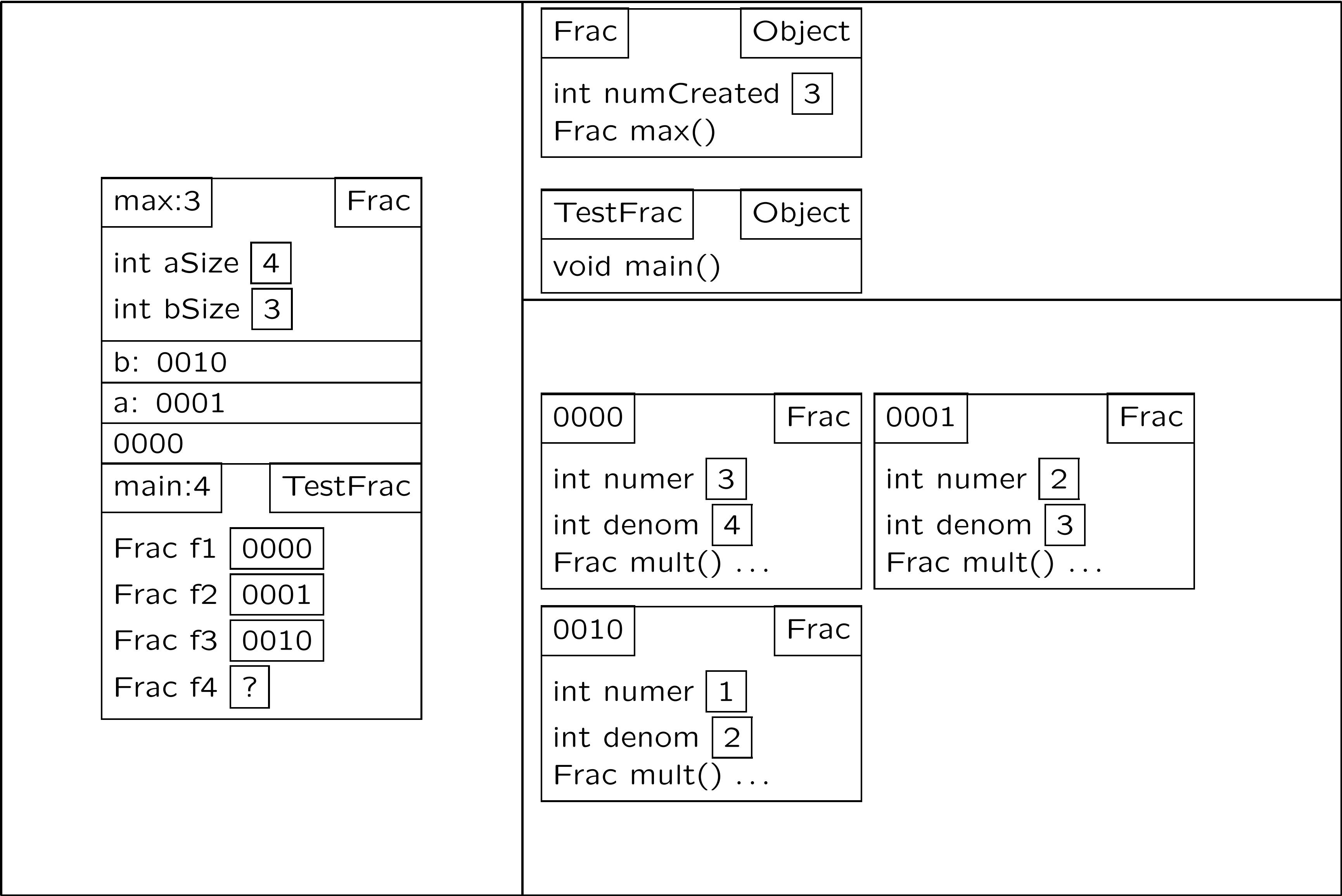
TestFrac Program

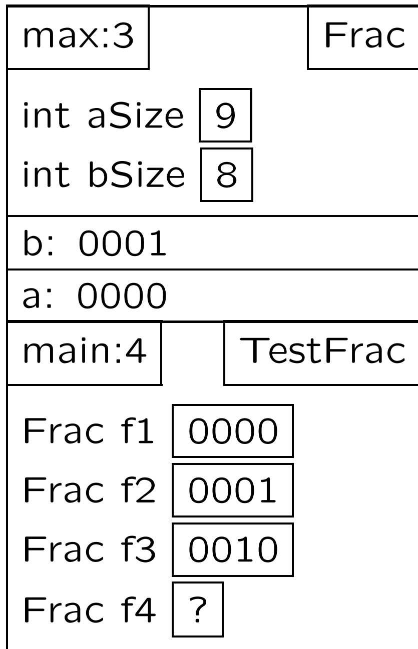
Now trace this fully.

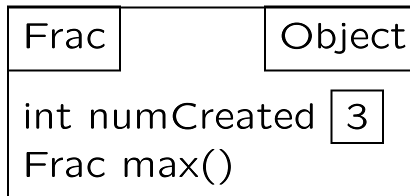
```
public class TestFrac {
    public static void main(String[] args) {
        Frac f1 = new Frac(3, 4);
        Frac f2 = new Frac(2, 3);
        Frac f3 = new Frac(1, 2);
        Frac f4 = Frac.max(f1, Frac.max(f2, f3));
    }
}

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;  }
}
```

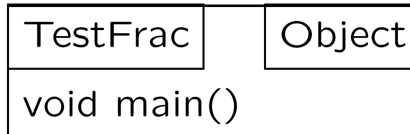


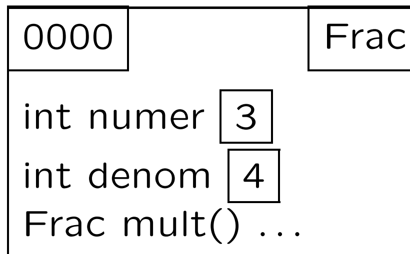
Frac
Object
int numCreated 3
Frac max()
TestFrac
Object
void main()
main:4
TestFrac
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 ?
0000
Frac
int numer 3
int denom 4
Frac mult() ...
0001
Frac
int numer 2
int denom 3
Frac mult() ...
0010
Frac
int numer 1
int denom 2
Frac mult() ...

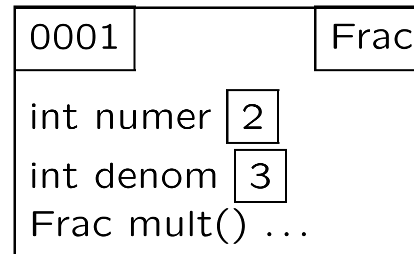

Frac
Object
int numCreated 3
Frac max()
TestFrac
Object
void main()
max:3
Frac
int aSize 4
int bSize 3
b: 0010
a: 0001
0000
main:4
TestFrac
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 ?
0000
Frac
int numer 3
int denom 4
Frac mult() ...
0001
Frac
int numer 2
int denom 3
Frac mult() ...
0010
Frac
int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

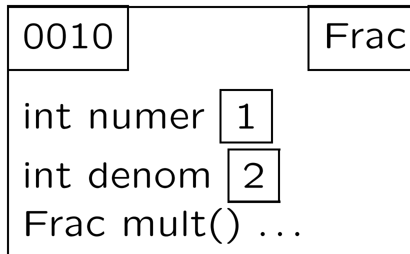

max:3
Frac
int aSize 9
int bSize 8
b: 0001
a: 0000
main:4
TestFrac
Frac f1 0000
Frac f2 0001
Frac f3 0010
Frac f4 ?

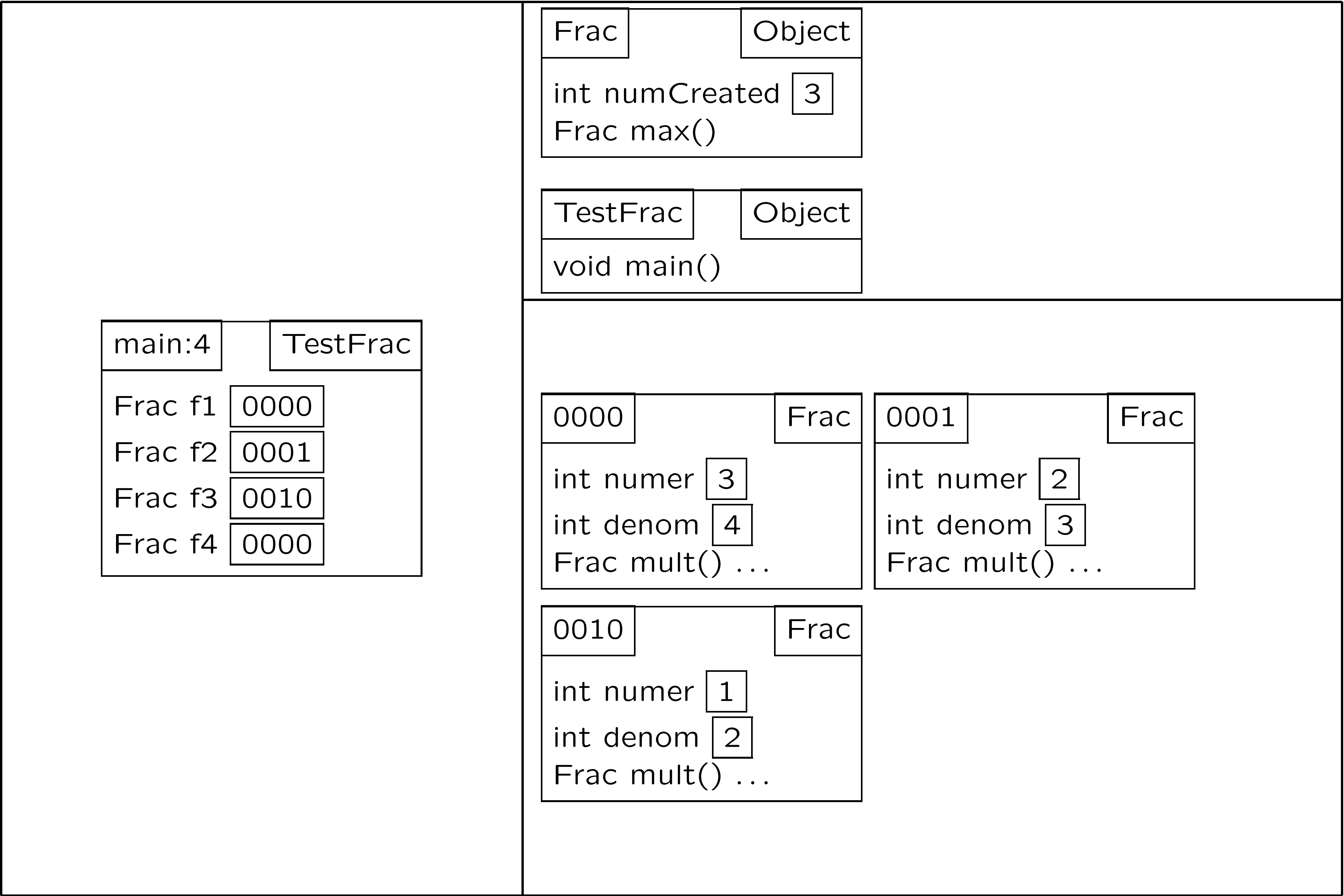

Frac
Object
int numCreated 3
Frac max()


TestFrac
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() ...



Frac
Object
int numCreated
3
Frac max()
TestFrac
Object
void main()
main:4
TestFrac
Frac f1
0000
Frac f2
0001
Frac f3
0010
Frac f4
0000
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() . . .