# 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 interface, class, object and running method has a separate region of memory to keep track of variable values and other related information.

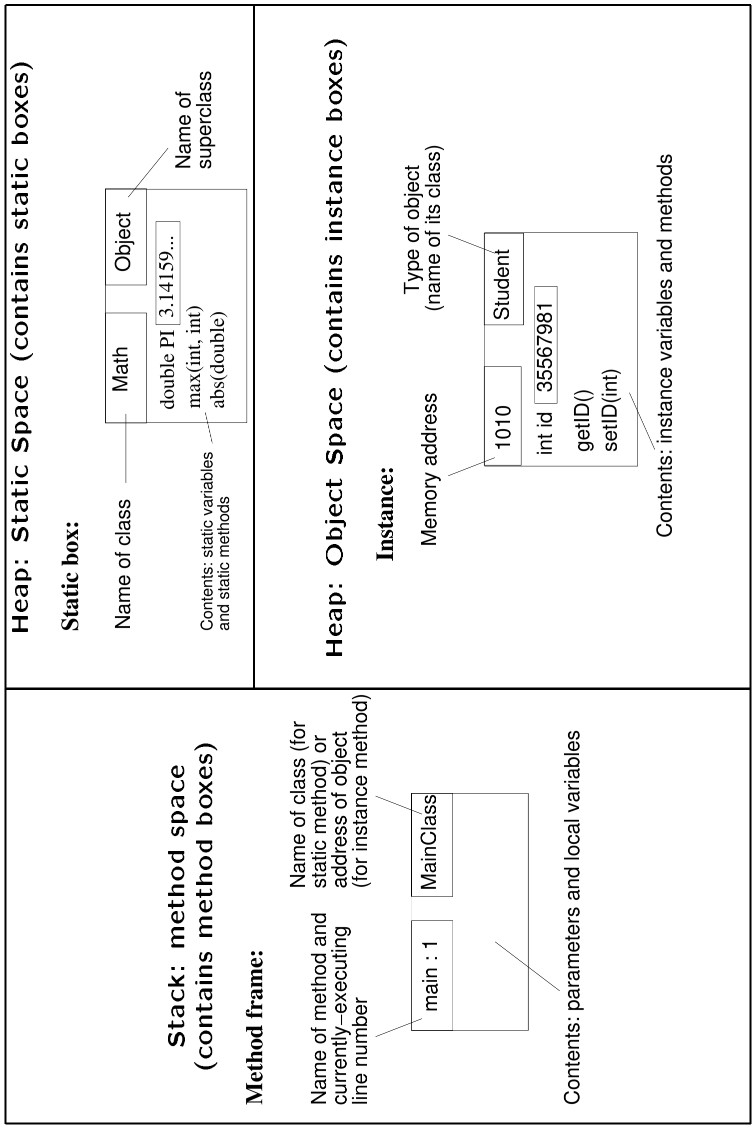
The program is executed line by line, with jumps between methods. Execution causes the stored information to change.

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


Static information (interface & class boxes) and instance information (object boxes) are stored in the *heap*.

Method information is stored in the *run-time stack*.





## Classes and interfaces

The source code for a class indicates:

- interfaces it implements
- the superclass it extends (if not explicitly indicated, then superclass is `Object`)
- its class members (`static` variables and methods), possibly with initial values
- constructors (if not explicitly indicated, then there's a default no-arg constructor)
- a template for objects of the class: instance members (non-`static` variables and methods)

The source code for an interface indicates:

- the interfaces it extends
- constants it defines (variables, automatically `public static final`)
- method headers (`public` instance) that implementing classes must provide



### Static space

For each class and interface mentioned in the program, draw a memory box in the static space. In each of these memory boxes record:

- the class/interface name [upper left corner]
- for a class, the superclass it extends [upper right corner]
- the list of interfaces (if any) implemented / extended [upper right corner]
- the names and values of static variables (if any) [in the main area of the box]; this includes all interface constants
- for a class, the signatures of the static methods (if any) [in the main area of the box]

Only one copy of each class and interface (and their static members) exists no matter how many objects are created.

Each variable starts with the default value for its type, unless it has an initializer. Record the values.



## Object space

An object is created by a `new` expression calling a constructor of some class. Draw a new memory box for it in the object space.

The object gets a unique address (different from all other objects in use), which we make up and write in the upper left corner. Represent the address as an arbitrary four digit hexadecimal number (e.g. `ABCD`, `0007`).

Divide the memory box for the object into a stack of boxes: the bottom-most part for the class of the object, and one part for each ancestor of that class.

The object gets a copy of the instance members of each of these classes. For each class, write the members in the corresponding part of the memory box and the name of the class in the upper right corner of the part.

Each variable starts with the default value for its type, unless it has an initializer. Record the values.

Execute the constructor as if it were an instance method called on the object through a reference of the same type as the object.

The constructor starts by executing the code in the superclass default constructor, unless the first statement specifically specifies how to start:

- `this(...)` selects another constructor of the class (based on the argument types)
- `super(...)` selects a superclass constructor (based on the argument types)

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



**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.

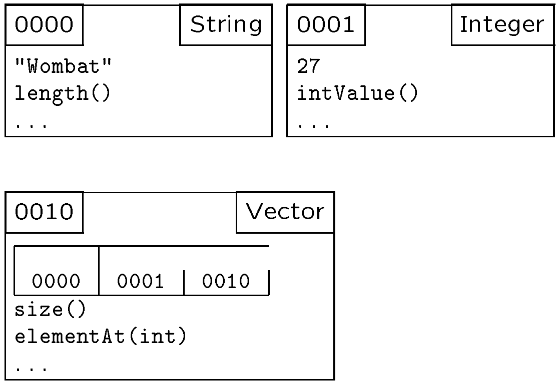
  Just make up a sensible notation. The next slide contains a few examples.

# 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:





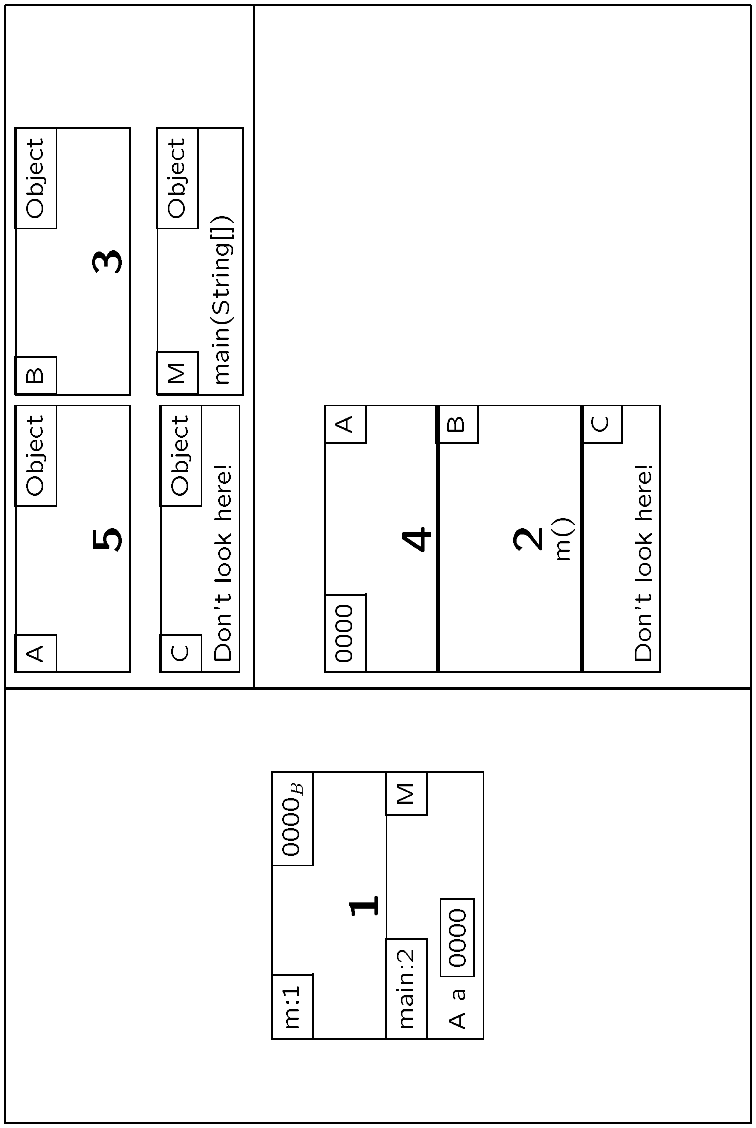

# Introduction to finding names

Consider this code:

```
public class A { ??? }

public class B extends A {
    ???
    public void m(???) {
        ???
        i = 3;   // Line 3, say.
    }
}

public class C extends B { ... }

public class M {
    public static void main(String[] args) {
        A a = new C();  a.m();
    }
}
```

The `???`'s indicate where `i` may have been declared. `i` may be a local variable or parameter in `m`, or a static or instance variable in `A` or `B`. (Or some combination!)

We need to find which variable `i` refers to, it's **target**.






## Finding variables and methods

After the static information is recorded, we're always tracing code inside the currently running method (at the top of the method stack). This method comes from a class or part of an object (recorded in the method's scope box).

Code within the method can *directly* refer to the local variables and parameters inside the method, and certain members. If the method comes from

- **a class `C`:** it can directly refer to the static members of the class and its ancestors, and the interfaces implemented by them.
- **the `C` part of an object:** it can directly refer to what it could if it were in class `C`, and also the instance members of the object's `C` part and ancestor parts.

Scope boxes serve as a guide for where to look next when looking for a name.

For a name not following a dot, look for it first (if it's a variable) as a local variable or parameter, then look "up" through the members the method can directly refer to, until you find it.

### Overriding

If the name refers to a `public` instance method, then finding the actual method that runs requires an extra step, due to overriding: look for the bottom-most version in the object.

### Dotted names: `e.name`

First determine `e`, and then pretend you're 'inside `e`' instead of inside the current method. If `e` is

- **a class name `C`:** act as if you're inside a method of class `C`.
- **a type `C` reference to an object:** act as if you're in the `C` part of the object.



## Overloading, Shadowing and Overriding

If two method declarations have the same name but different parameter types, we say the name is **overloaded**. When looking for one of these methods use its signature; the other method doesn't affect the search.

If a variable or a method with the same signature is redeclared in a subclass we say it **shadows** the declaration in the superclass, except in the case of overriding. With shadowing, which member is accessed depends on the type used to reference it (i.e., in which class or part of object you start looking for it).

## More about Overriding

Overriding allows us to customize the behaviour of a `public instance` method based on the type of object (as opposed to the type used to reference it).

Why not have something like overriding occur in other situations?

- `private` **instance method:** a subclass could inadvertently (its programmer shouldn't have to know about *private* ancestor methods) change its behaviour
- **variable:** should be private; it's also unclear what it means to customize a variable
- `static` **member:** considered to belong to a class/interface, and it's not possible to refer to a class/interface without knowing exactly which one



## `this`

In a running instance method, the object part it came from is recorded in the scope box as an address and type, and can be referred to as `this`.

One use of `this` is to force variable lookup to skip the parameters and local variables.

Trace the following example ...

```
public class TestThis {
    public static void main(String[] args) {
        (new TestThis()).test();
    }
    private int v = 1;
    public void test() {
        int v = 2;
        System.out.println(this.v);
    }
}
```

`this` is also used when passing a reference to this object to another method, and when checking whether some reference refers to this object.

## Casting

Casting changes the type of an expression. We can cast an object reference to the type of any part of the object, or to any interface the parts implement.

The main use of casting is at compile time to reassure the compiler that we can do certain things with a reference.

But it does have two effects when the program is running:

- since it changes the type, it changes where lookup starts
- if the reference refers to an object that doesn't have the part cast to, an exception occurs

Keep in mind that casting doesn't change the object in any way. It just temporarily changes how we're referring to it.



## `super`

In a running instance method, the object part immediately above `this` can be referred to as `super`. We could also refer to that part by casting `this`. So why bother with `super`?

Because `super` has a special behaviour: it prevents overriding (and is the one exception to our member lookup rules). It's often used inside an overriding method to include the behaviour of the overriden method.

Trace the following example ...

## `super` Example

```
public class TestSuper {
    public static void main(String[] args) {
        (new Sub()).m1();
        System.out.println("---");
        (new Sub()).m2();
    }
}
class Super {
    void m1() {
        System.out.println("Super!");
    }
}
class Sub extends Super {
    void m1() {
        super.m1();
        System.out.println("And Sub!");
    }
    void m2() {
        ((Super)this).m1();
    }
}
```



## Variables

**Local variable declaration:** `Type var;`

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

**Assignment:** `var = rhs;`

1. Evaluate the right side.
2. Find the target of the variable on the left side.
3. Write the value of the right side in the box for the variable.

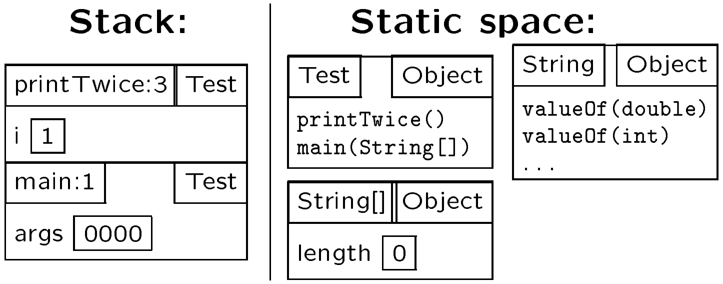
**Initialization:** `Type var = rhs;`

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

# Run-time stack example

```
public class Test {
    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.)






## 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() );
```

# Method call

### Arguments

1. Evaluate them in order: inside out, left to right.
2. Put their values into boxes on top of the stack: they'll be 'picked up' by the method as parameters.

### Executing the method

1. Find the target of the method.
2. Draw a frame for the method on top of the stack, including the argument boxes that will already be there.
3. Write the method name in the upper left corner, and where the method was found (address + part, or class) in the upper right corner.
4. Name the argument boxes to their corresponding paramater names.
5. Indicate the line number about to be executed, by writing `:1` after the method name.
6. Execute the body of the method line by line, incrementing the line number each time a line is finished.



### Returning: `return expr;`

1. Evaluate `expr` and replace the current method frame with the value.
2. If the value is being used as an argument, then it's ready to be picked up. Otherwise, use it and remove it.

## 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).

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

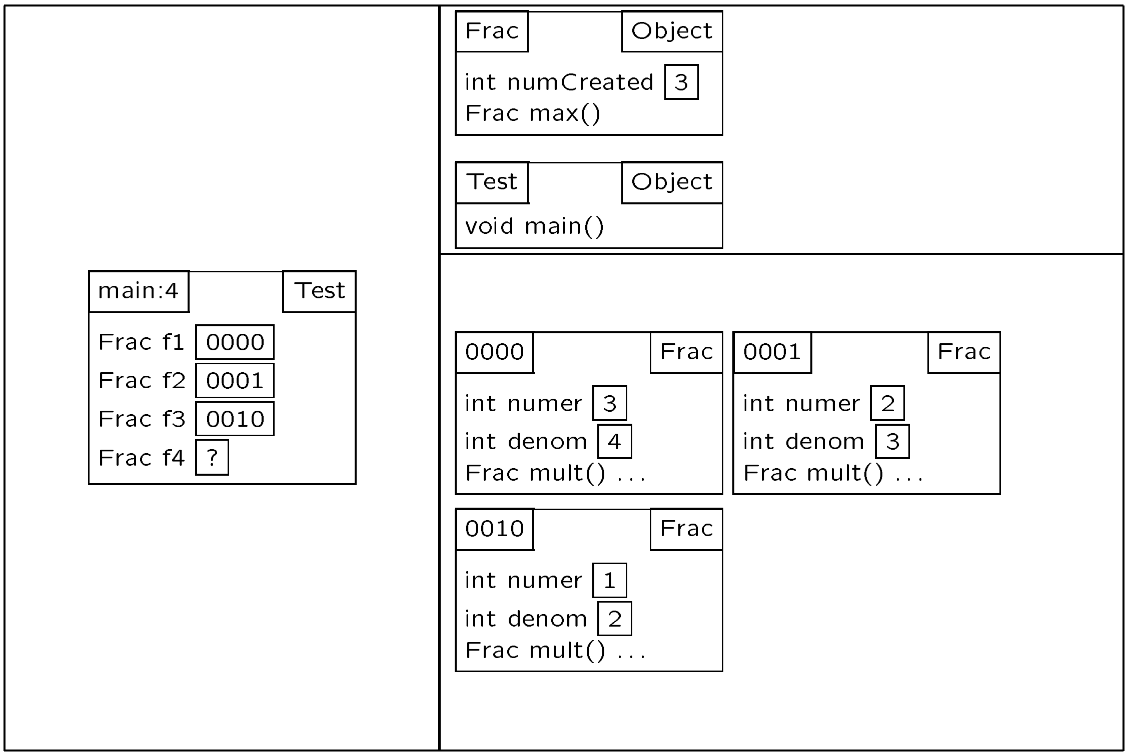


## Practice

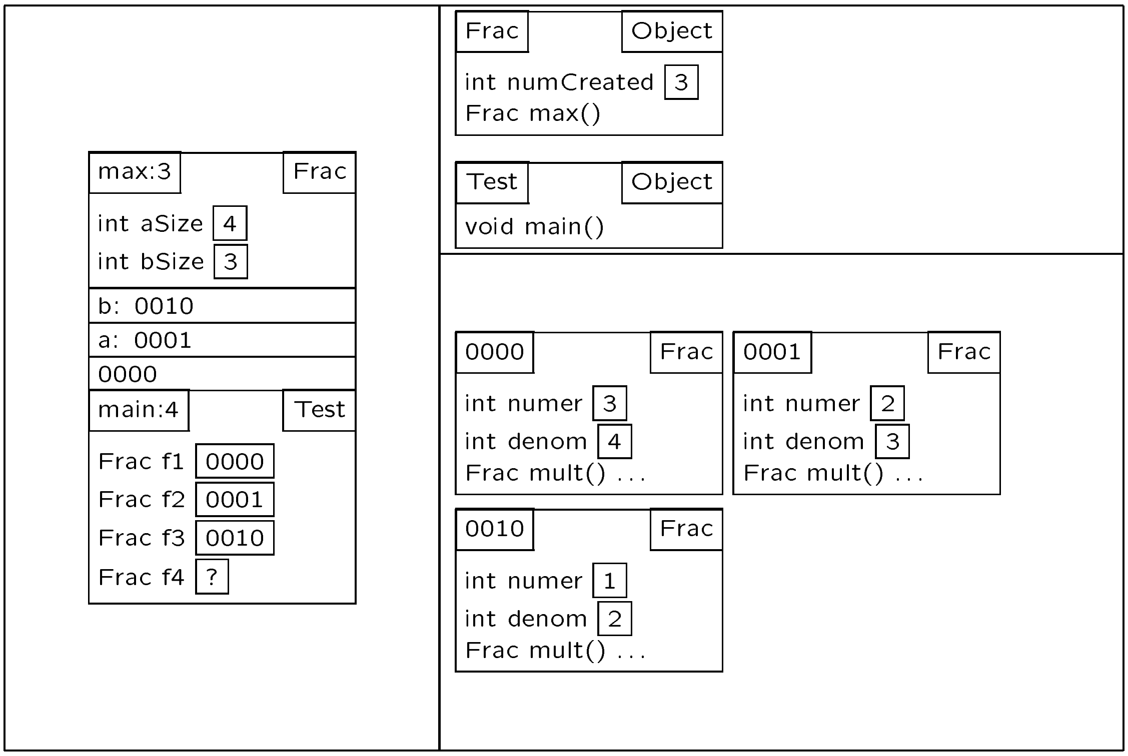
Trace this.

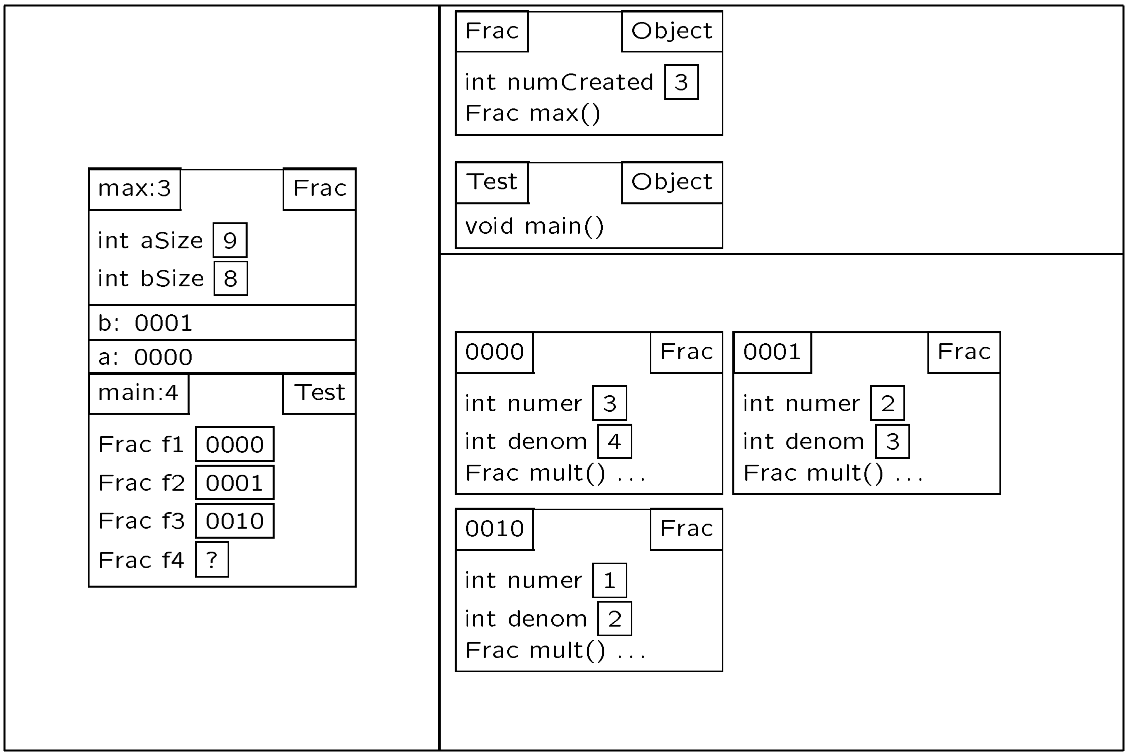
```
public class TestFrac {
    public static void main(String[] args) {
        Frac f1 = new Frac(3, 4);
        Frac f2 = new Frac(2, 3);
        Frac f3 = new Frac(1, 2);
        Frac f4 = Frac.max(f1, Frac.max(f2, f3));
    }
}

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;  }
}
```

28


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() . . .

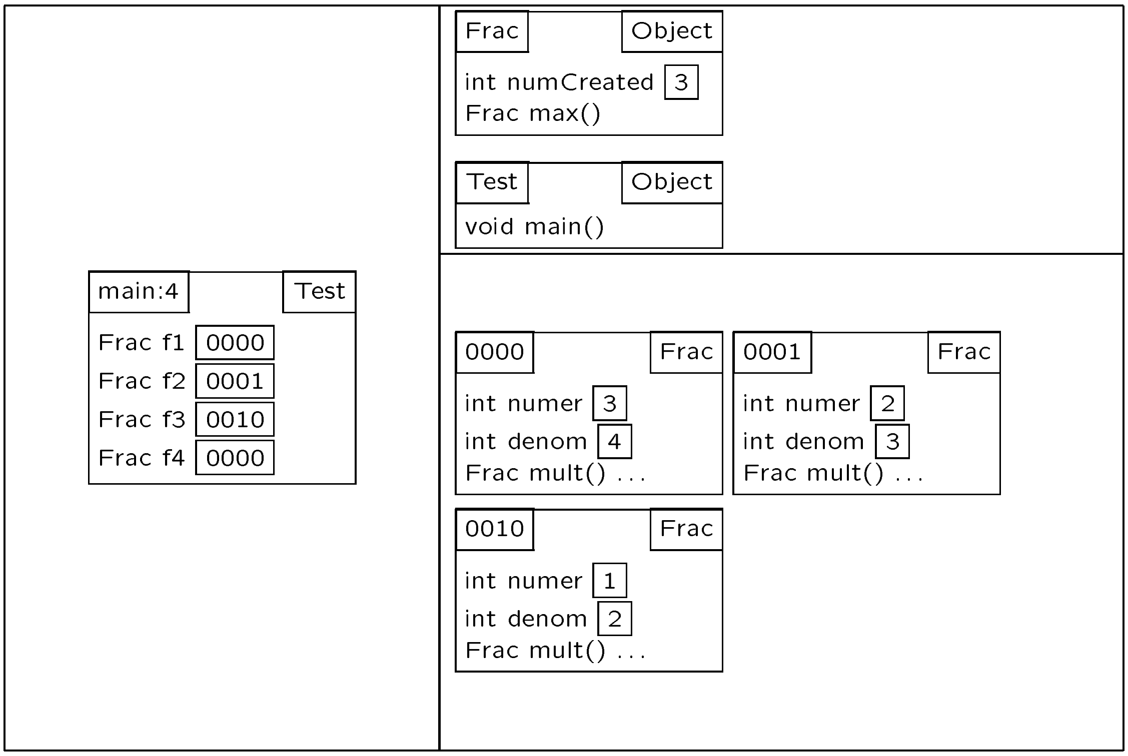
29


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 ?
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() ...


30



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() ...


31

# More Practice

```
class A {
  public static int k;
  public int i;
  public void m(int i) {
    this.i = i;
    f();
  }
  public void f() {
    System.out.println(
      "A" + i);
  }
}
```

```
class B extends A {
  public static int k;
  public int i;
  public void f() {
    System.out.println(
      "B" + i);
    super.f();
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    A ab = new B();
    a.m(3);
    b.m(5);
    ab.m(-2);
    a.k = 12;
    ab.k = 15;
    b.k = 4;
    ((A)b).k = 8;
    ((B)ab).k = 4;
    ((B)a).k = 1;
  }
}
```