The target of a reference

Consider this code:

```
public class A { ??? } // 4. Variable in class A?
public interface I {
  static final int CNST = 66;
  ???   // 3. Constant in I?
}
public class B implements I extends A {
  ???                  // 2. Variable in class B?
  public void m(???) {      // 1. Parameter?
    ???                     // 1. Local variable?
    System.out.println(i);  // Line 3, which i?
  }
}
public class C extends B { }
public class M {
  public static void main(String[] args) {
    B b = new C();  b.m();
    I i = b; ...
  }
}
```

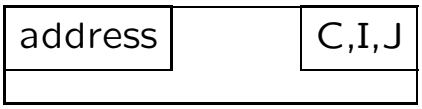
The target of `i` needs to be found. The *target* of an identifier is the variable or method to which it refers. The ???'s indicate where `i` may have been declared.

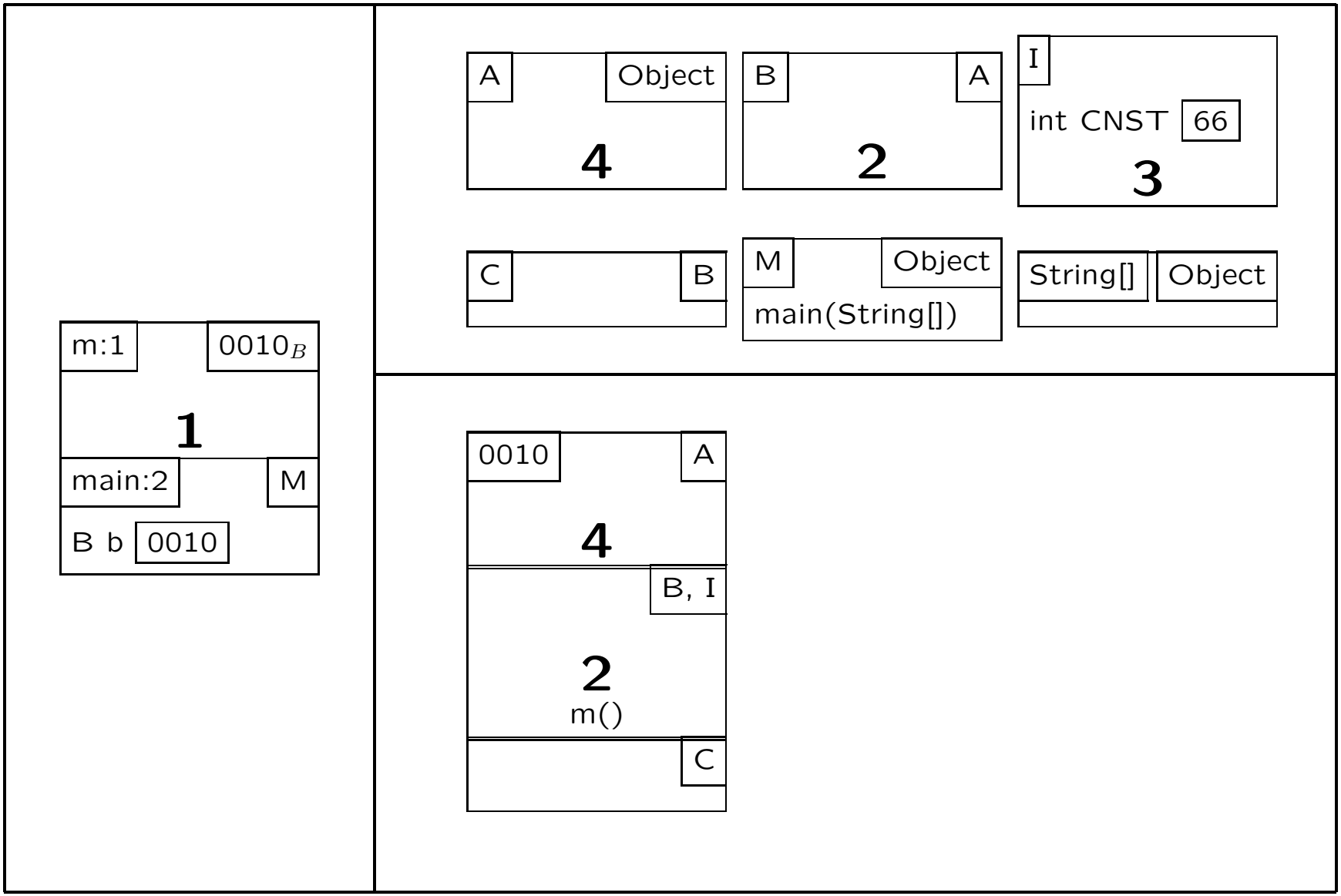
Interfaces in the Memory Model

Each `interface` has a static box to contain its `static` constants (if any). These boxes look just like class boxes.

In objects, interfaces don't act like full-fledged classes. If a class `C` implements an interface (or several) then write a comma-separated list of those interfaces in an object of that class.

For example, if `C` implements `I` and `J`, then an object of class `C` looks like this:




m:1
0010$_B$
1
main:2
M
B b
0010
A
Object
4
B
A
2
I
int CNST
66
3
C
B
M
Object
main(String[])
String[]
Object
0010
A
4
B, I
2
m()
C

Two-Fingered Look-Up

The preceding memory model illustrates the **two-fingered look-up algorithm**, where one finger points at a subpart of the object and the other points at the corresponding static box.

If we do not find the target at some stage of this procedure then where, exactly, do we look next?

Answer: We look upwards in the next enclosing scope.

To understand what this means we first need to specify the notion of a declaration's scope.

Scope of Local Variables and Parameters

Programs are written in nested blocks of code, often enclosed by braces "{" and "}" (eg. around the method body itself, an if-block, or loop).

Declarations of local variables in such a block persist for the remainder of the block, including any nested sub-blocks. This region of code is called the **scope** of the declaration.

The scope of a method parameter is the method body, including any nested sub-blocks.

A variable declared outside a method may be "hidden" by a parameter or local variable of the same name declared within the method.

Special case: Variables can be declared in the for statement (i.e., `for (int k=0; ...) {...}`), and their scope is then that statement together with the loop body.

Scope of Class and Interface Members

The body of a class definition also forms a block of code. The scope of members declared in a class are as follows:

- static member: its whole class, including all the constructors and methods, and the nested scopes of any sub-classes.
- instance member: the constructors and instance methods of its particular object, and the nested scopes from the sub-class parts of the same object.

Similarly, the scope of static constants declared in an interface include any class `C` that is declared to implement that interface. As always, this scope includes the nested scopes of any sub-classes.

Enclosing scope

The *enclosing scope* of a method, object, or class is the next-bigger scope. The enclosing scopes of various memory boxes are:

- static method on run-time stack: the static box it is defined in.
- instance method on run-time stack: the box of the object its method body is defined in, along with the static box for its class.
- static box for a class: any interface that this class declares to implement, followed by the static space for the superclass.
- subclass box in an object: (note that the static space for this subclass is included as part of this scope) the enclosing scope includes any interfaces that the subclass declares to implement, followed by the superclass part of the object and the corresponding static box for the superclass.

The Dot Operator

For a bare reference (eg. `ident`) we start searching for the target in the current scope. (Here and in what follows we use the text "`ident`" to represent either a variable name or a method call.)

To start the search in some other box we can use a dot operator (eg. `box.ident`).

Here the type and value of "`box`" specifies the particular box in the memory model in which to start looking for the target `ident`.

Just as for a bare reference, if the target is not found then the search continues, from the specified box, upwards through the enclosing scopes.

Dot Operator Examples

For example, consider the situation depicted on p. 23 and p. 25 above.

In the `main` method, the variable `b` is of type `B` (a class) and refers to an object at address `x0010`. This object is actually of class `C`, a subclass of `B`. The variable `i` in this method has interface type `I`. After the assignment statement is executed, it will also have the value `x0010`.

Finally, note class `B` implements `I` and `B` is a subclass of `A`.

Expression	Box Type	Box Value	Named Memory Box
`b.ident`	`B`	`x0010`	`B` subbox at `x0010`.
`B.ident`	`B`	n/a	Static box of class `B`.
`I.CNST`	`I`	n/a	Static box of interface `I`.
`i.m()`	`I`	`x0010`	Bottom subbox at `x0010`.
`((A) b).ident`	`A`	`x0010`	`A` subbox at `x0010`.
`((C) b).ident`	`C`	`x0010`	`C` subbox at `x0010`.
`((I) b).m()`	`I`	`x0010`	Bottom subbox at `x0010`.

Casting

An expression of the form "`(A) b`" *casts* `b` to a reference of type `A`. Several casts appear in the table on the previous page.

Casting changes the type of an expression but **does not change the address of an object, nor the class of an object**. Casting can be used with the dot operator to specify a particular starting box for target look-up.

We can make assignments to higher and more general types of variables without casting. These are called widening casts.

For the same example as on the previous page:

```
A a = b; // Widening cast
I i = b; // Widening cast
C c = b; // Not valid.
C c = (C) b; // Valid narrowing cast.
M m = (M) b; // Not valid.
```

Java reference lookup algorithm

To find the target of a reference:

1. Start looking in the box named by the reference.

2. Look upwards through the enclosing scopes until you first find the target.

3. If the first target found is a non-private instance method, then repeat step 2, this time starting at the bottom-most box in the current object.

An example of Steps 1 and 2 above is given by the two-fingered look-up of the variable `i`, as illustrated on p. 25.

Step 3 of the algorithm is called dynamic method look-up, and it implements *method overriding*.

Matching Methods

When looking for a method, check whether any method signature matches the call. The method signature consists of both the method name and the parameter types. You can use the same method name for several methods if the parameter lists are different. (This is "overloading", not "overriding".)

The return type is *not* part of the signature.

A declared method `C m(R r)` matches the method call `b.m(p)` if the variable `p` can be assigned to a variable of type `R` (with a widening cast or no cast).

Shadowing and overriding

If there are several instance methods with matching signatures in the subparts of an object, and all are non-private, then the bottom-most method body in the object is the target: that method *overrides* the others.

Because of overriding, finding the target of a non-private instance method adds an extra step to the look-up algorithm.

A variable declared in a subclass is said to *shadow* a variable of the same name declared in a superclass. Note the third step of the look-up algorithm is never used for variables.

Variables are always shadowed, not overridden. Unlike overridden methods, we can always access (public) shadowed variables by casting.

What the heck does this print?

```
class Tricky {
    public static void main(String[] s) {
        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;
  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;
  }
}
```

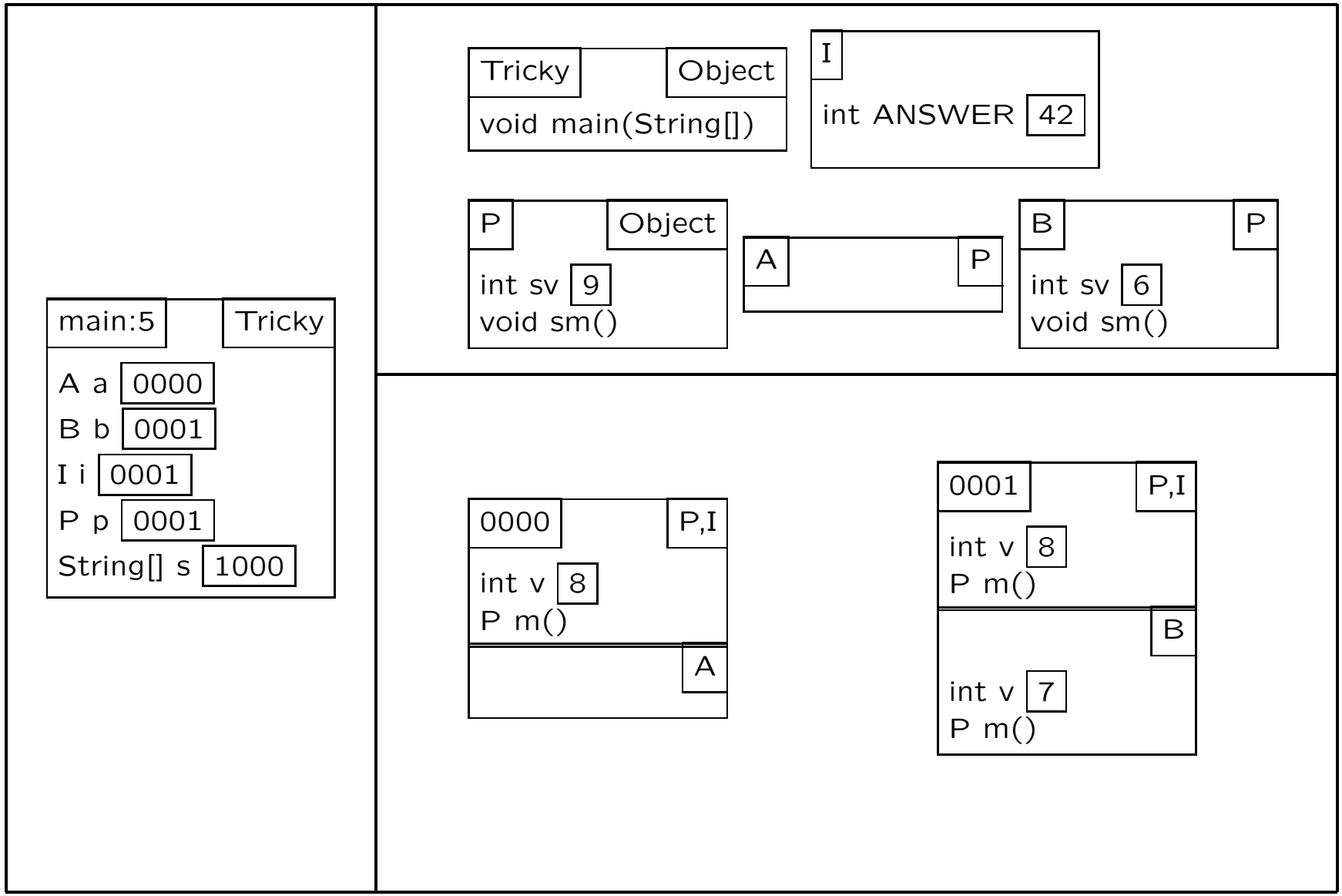
```
// Sibling classes A and B

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;
  }
}
```


main:5
Tricky
A a 0000
B b 0001
I i 0001
P p 0001
String[] s 1000
Tricky
Object
void main(String[])
I
int ANSWER 42
P
Object
int sv 9
void sm()
A
P
B
P
int sv 6
void sm()
0000
P,I
int v 8
P m()
A
0001
P,I
int v 8
P m()
B
int v 7
P m()

Targets in our Tricky Program

Remember that to find the target of expressions `r.v` and `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`	`0001`	`v` in `B` at `0001`	7
`a.v`	`A`	`0000`	`v` in `P` at `0000`	8
`((P) b).v`	`P`	`0001`	`v` in `P` at `0001`	8
`b.sv`	`B`	`0001`	`sv` in `B`	6
`a.sv`	`A`	`0000`	`sv` in `P`	9
`((P) b).sv`	`P`	`0001`	`sv` in `P`	9

Expression	Type of r	Value of r	Target method
`b.m()`	`B`	`0001`	`m()` in `B` at `0001`
`a.m()`	`A`	`0000`	`m()` in `P` at `0000`
`((P) b).m()`	`P`	`0001`	`m()` in `B` at `0001`
`((I) b).m()`	`I`	`0001`	`m()` in `B` at `0001`
`b.sm()`	`B`	`0001`	`sm()` in `B`
`a.sm()`	`A`	`0000`	`sm()` in `P`
`((P) b).sm()`	`P`	`0001`	`sm()` in `P`

Here we assume these expressions appear within the `main` method of the Tricky example on pp. 36-37).

Java Naming Conventions

Following the Java naming conventions helps to make programs less confusing and more readable. (Your marker will be happier.) They include:

- Class and interface names begin with a capital.
- Constants are all capitals.
- Variables and methods begin with lowercase, and have descriptive names.
- When a dot operator is used to reference a static item, use the correct target class or interface name (e.g., `C.ident` or `I.CNST`).

Exercise: Rewrite `Tricky` following these conventions where possible (finding descriptive names here may be tough).

Keywords `this` and `super`

References `this.member` and `super.member` in a method body `m()`:

- `this`: names the memory box containing the *instance* method body `m()` (the use of `this` within a static method is illegal).
- `super`: names the memory box corresponding to the parent class for the box containing the method body `m()`.

When `super` is used, the dynamic method look-up (i.e. step 3 of the Java reference look-up algorithm) is *not* performed. So we can use `super` to get at an overridden method in an ancestor class.

Trace the following examples ...

This super example

Suppose we modify the Tricky example on p. 36 as follows:

```
// Suppose class B had this additional method:
public void newMethod() {
    int v = 99;
    super.m();
    System.out.println("v = " + v);
    System.out.println("this.v = " + this.v);
}

// 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);
    }
}
```

this() and super()

The same keywords can be used in a different way, but only in **constructors**:

- `this()` refers to the no-parameter constructor within the current class.
- `super()` refers to the parent class's no-parameter constructor
- Moreover `super(10)` refers to the parent class's constructor which takes one int parameter. Similarly for multi-parameter constructors.
- Both `this( ... )` and `super( ... )` can ONLY appear on the first line of the constructor.
- If `this( ... )` or `super( ... )` does NOT appear on the first line of the constructor, then super() is added implicitly. Beware this will cause a compile time error if the parent class does not have a no-parameter constructor.