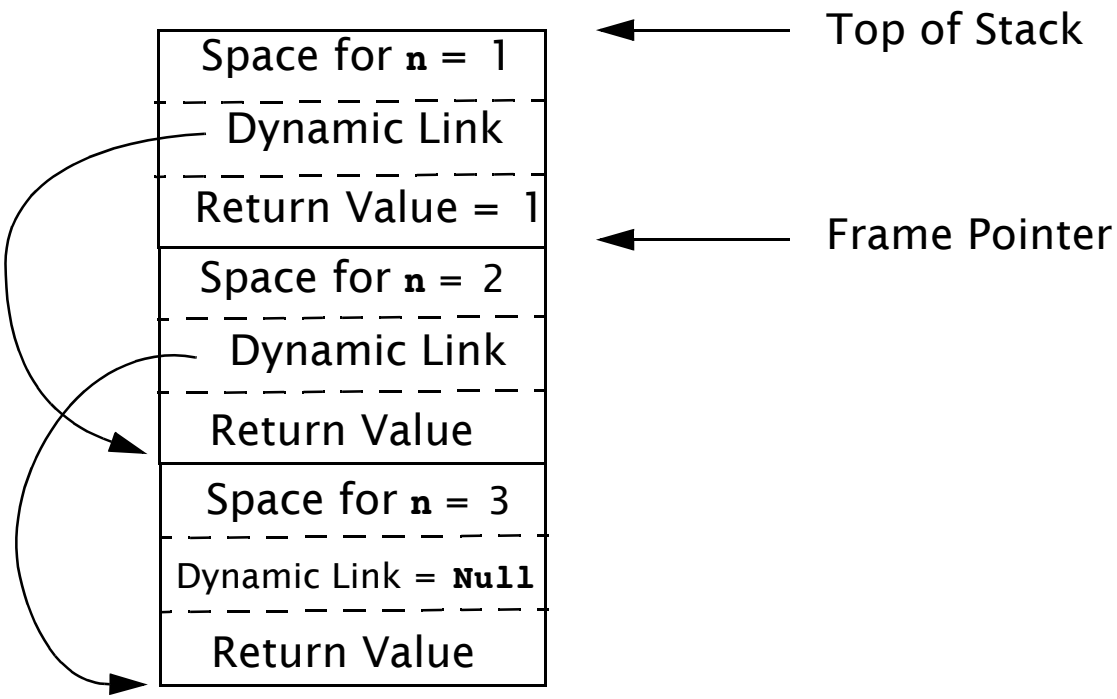
Dynamic Links

Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common to save the caller's frame pointer as part of the callee's control information.

Each frame points to its caller's frame on the stack. This pointer is called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor.

The run- time stack corresponding to a call of **`fact(3)`**, with dynamic links included, is:



Classes and Objects

C, C++ and Java do not allow procedures or methods to nest.

A procedure may not be declared within another procedure.

This simplifies run- time data access—all variables are either global or local.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Java and C++ allow classes to have *member functions* that have direct access to instance variables.

Consider:

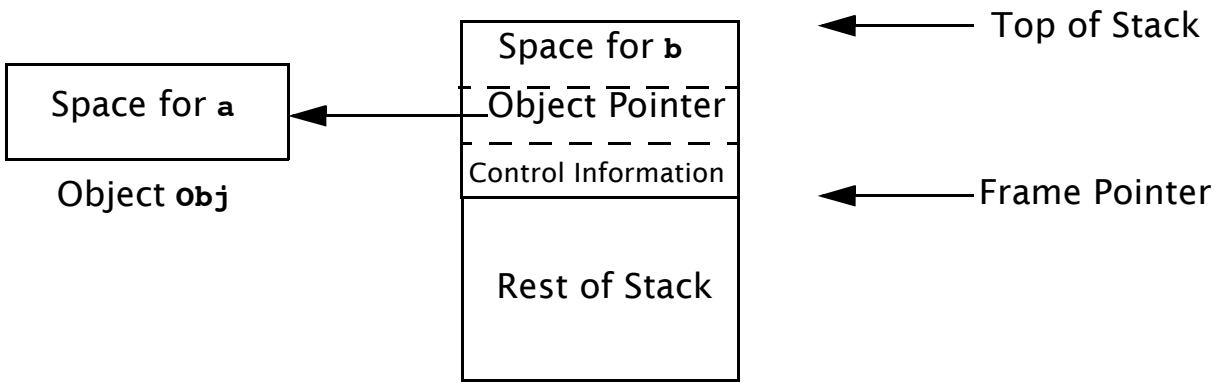
```
class K {
  int a;
  int sum(){
  int b;
  return a+b;
} }
```

Each object that is an instance of class **K** contains a member function **sum**. Only one translation of **sum** is created; it is shared by all instances of **K**.

When **sum** executes it needs *two* pointers to access local and object- level data.

Local data, as usual, resides in a frame on the run- time stack.

Data values for a particular instance of **`K`** are accessed through an object pointer (called the **`this`** pointer in Java and C++). When **`obj.sum()`** is called, it is given an extra *implicit parameter* that a pointer to **`obj`**.



When **`a+b`** is computed, **`b`**, a local variable, is accessed directly through the frame pointer. **`a`**, a member of object **`obj`**, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow inheritance via subclassing. A new class can extend an existing class, adding new fields and adding or redefining methods.

A subclass **`D`**, of class **`C`**, maybe be used in contexts expecting an object of class **`C`** (e.g., in method calls).

This is supported rather easily—objects of class **`D`** always contain a class **`C`** object within them.

If **`C`** has a field **`F`** within it, so does **`D`**. The fields **`D`** declares are merely *appended* at the end of the allocations for **`C`**.

As a result, access to fields of **`C`** within a class **`D`** object works perfectly.

Jump Code

The JVM code we generate for the following if statement is quite simple and efficient.

```
if (B)
  A = 1;
else
  A = 0;
```

```
    iload 2  ; Push local #2 (B) onto stack
    ifeq L1  ; Goto L1 if B is 0 (false)
    iconst_1 ; Push literal 1 onto stack
    istore 1 ; Store stk top into local #1(A)
    goto L2  ; Skip around else part
L1: iconst_0 ; Push literal 0 onto stack
    istore 1 ; Store stk top into local #1(A)
L2:
```

In contrast, the code generated for

```
if (F == G)
  A = 1;
else
  A = 0;
```

(where F and G are local variables of type integer)

is significantly more complex:

```
  iload 4       ; Push local #4 (F) onto stack
  iload 5       ; Push local #5 (G) onto stack
  if_icmpeq L1  ; Goto L1 if F == G
  iconst_0      ; Push 0 (false) onto stack
  goto L2       ; Skip around next instruction
L1:
  iconst_1      ; Push 1 (true) onto the stack
L2:
  ifeq L3       ; Goto L3 if F==G is 0 (false)
  iconst_1      ; Push literal 1 onto stack
  istore 1      ; Store top into local #1(A)
  goto L4       ; Skip around else part
L3:
  iconst_0      ; Push literal 0 onto stack
  istore 1      ; Store top into local #1(A)
L4:
```

The problem is that in the JVM relational operators don't store a boolean value (0 or 1) onto the stack. Rather, instructions like **`if_icmpeq`** do a *conditional branch*.

So we branch to a push of 0 or 1 just so we can test the value and do a *second* conditional branch to the else part of the conditional.

Why did the JVM designers create such an odd way of evaluating relational operators?

A moment's reflection shows that we rarely actually *want* the value of a relational or logical expression. Rather, we usually

only want to do a conditional branch based on the expression's value in the context of a conditional or looping statement.

Jump code is an alternative representation of boolean values. Rather than placing a boolean value directly on the stack, we generate a conditional branch to either a **true label** or a **false label.** These labels are defined at the places where we wish execution to proceed once the boolean expression's value is known.

Returning to our previous example, we can generate **F==G** in jump code form as

```
iload 4       ; Push local #4 (F) onto stack
iload5        ; Push local #5 (G) onto stack
if_icmpne L1  ; Goto L1 if F != G
```

The label **L1** is the "false label." We branch to it if the expression **F == G** is false; otherwise, we "fall through," executing the code that follows. We can then generate the then part, defining **L1** at the point where the else part is to be computed. The code we generate is:

```
  iload 4       ; Push local #4 (F) onto stack
  iload5        ; Push local #5 (G) onto stack
  if_icmpne L1  ; Goto L1 if F != G
  iconst_1      ; Push literal 1 onto stack
  istore 1      ; Store top into local #1(A)
  goto L2       ; Skip around else part
L1:
  iconst_0      ; Push literal 0 onto stack
  istore 1      ; Store top into local #1(A)
L2:
```

This instruction sequence is significantly shorter (and faster) than our original translation. Jump code is routinely used in ifs, whiles and fors where we wish to alter flow- of- control rather than compute an explicit boolean value.

Jump code comes in two forms, **`JumpIfTrue`** and **`JumpIfFalse`**.

In **`JumpIfTrue`** form, the code sequence does a conditional jump (branch) if the expression is true, and "falls through" if the expression is false. Analogously, in **`JumpIfFalse`** form, the code sequence does a conditional jump (branch) if the expression is false, and "falls through" if the expression is true. We have two forms because different contexts prefer one or the other.

It is important to emphasize that even though jump code looks unusual, it is just an alternative representation of boolean values. We can convert

a boolean value on the stack to jump code by conditionally branching on its value to a true or false label.

Similarly, we convert from jump code to an explicit boolean value, by placing the jump code's true label at a load of 1 and the false label at a load of 0.

Short-Circuit Evaluation

Our translation of the `&&` and `||` operators parallels that of all other binary operators: evaluate both operands onto the stack and then do an "and" or "or" operation.

But in C, C++, C#, Java (and most other languages), `&&` and `||` are handled specially.

These two operators are defined to work in "short circuit" mode. That is, if the left operand is sufficient to determine the result of the operation, the right operand *isn't evaluated*.

In particular **`a&&b`** is defined as **`if a then b else false`**.

Similarly **a||b** is defined as **if a then true else b**.

The conditional evaluation of the second operand isn't just an optimization—it's essential for correctness. For example, in **(a!=0)&&(b/a>100)** we would perform a division by zero if the right operand were evaluated when **a==0**.

Jump code meshes nicely with the short- circuit definitions of && and ||, since they are already defined in terms of conditional branches.

In particular if **exp1** and **exp2** are in jump code form, then we need generate *no further code* to evaluate **exp1&&exp2**.

To evaluate **`&&`**, we first translate **`exp1`** into **`JumpIfFalse`** form, followed by **`exp2`**. If **`exp1`** is false, we jump out of the whole expression. If **`exp1`** is true, we fall through to **`exp2`** and evaluate it. In this way, **`exp2`** is evaluated only when necessary (when **`exp1`** is true).

Similarly, once **`exp1`** and **`exp2`** are in jump code form, **`exp1||exp2`** is easy to evaluate. We first translate **`exp1`** into **`JumpIfTrue`** form, followed by **`exp2`**. If **`exp1`** is true, we jump out of the whole expression. If **`exp1`** is false, we fall through to **`exp2`** and evaluate it. In this way, **`exp2`** is evaluated only when necessary (when **`exp1`** is false).

As an example, let's consider

```
if ((A>0)||(B<0 && C==10))
  A = 1;
else
  A = 0;
```

Assume **A**, **B** and **C** are all local integers, with indices of 1, 2 and 3 respectively.

We'll produce a **JumpIfFalse** translation, jumping to label **F** (the else part) if the expression is false and falling through to the then part if the expression is true.

Code generators for relational operators can be easily modified to produce both kinds of jump code—we can either jump if the relation holds

(**JumpIfTrue**) or jump if it doesn't hold (**JumpIfFalse**). We produce the following JVM code sequence which is quite compact and efficient.

```
    iload 1      ; Push local #1 (A) onto stack
    ifgt L1      ; Goto L1 if A > 0 is true
    iload 2      ; Push local #2 (B) onto stack
    ifge F       ; Goto F if B < 0 is false
    iload 3      ; Push local #3 (C) onto stack
    bipush 10    ; Push a byte immediate (10)
    if_icmpne F  ; Goto F if C != 10
L1:
    iconst_1     ; Push literal 1 onto stack
    istore 1     ; Store top into local #1(A)
    goto L2      ; Skip around else part
F:
    iconst_0     ; Push literal 0 onto stack
    istore 1     ; Store top into local #1(A)
L2:
```

First **A** is tested. If it is greater than zero, the control expression must be true, so we skip the rest of the expression and execute the then part.

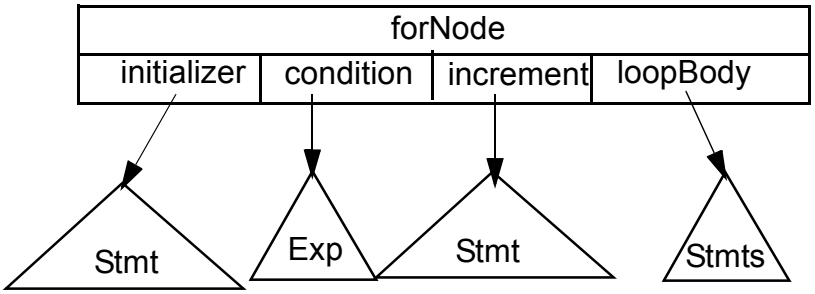
Otherwise, we continue evaluating the control expression.

We next test **B**. If it is greater than or equal to zero, **B<0** is false, and so is the whole expression. We therefore branch to label **F** and execute the else part.

Otherwise, we finally test **C**. If **C** is not equal to 10, the control expression is false, so we branch to label **F** and execute the else part.

If **C** is equal to 10, the control expression is true, and we fall through to the then part.

For Loops



For loops are translated much like while loops.

The AST for a for loop adds subtrees corresponding to loop initialization and increment.

For loops are expected to iterate many times. Therefore after executing the loop initialization, we skip past the loop body and increment code to reach the termination

condition, which is placed at the bottom of the loop.

```
 {Initialization code}
 goto   L1
L2:
 {Code for loop body}
 {Increment code}
L1:
 {Condition code}
 ifne  L2  ; branch to L2 if true
```

```
cg(){ // for forLoopNode
   String skip = genLab();
   String top = genLab();
   initializer.cg();
   branch(skip);
   defineLab(top);
   loopBody.cg();
   increment.cg();
   defineLab(skip);
   condition.cg();
   branchNZ(top);
}
```

As an example, consider this loop (`i` and `j` are locals with variable indices of 1 and 2)

```
for (i=100;i!=0;i--) {
     j = i;
}
```

The JVM code we generate is

```
  bipush 100   ; Push 100
  istore 1     ; Store into #1 (i)
  goto L1      ; Skip to exit test
L2:
  iload 1      ; Push local #1 (i)
  istore 2     ; Store into #2 (j)
  iload 1      ; Push local #1 (i)
  iconst_1     ; Push 1
  isub         ; Compute i-1
  istore 1     ; Store i-1 into #1 (i)
L1:
  iload 1      ; Push local #1 (i)
  ifne L2      ; Goto L2 if i is != 0
```

Java, C# and C++ allow a local declaration of a loop index as part of initialization, as illustrated by the following for loop

```
for (int i=100; i!=0; i--) {
  j = i;
}
```

Local declarations are automatically handled during code generation for the initialization expression. A local variable is declared within the current frame with a scope limited to the body of the loop. Otherwise translation is identical.